UiO : Department of Informatics
University of Oslo

Interactive Zooming and Panning in Panoramic Video Using WebGL

Mattias Håheim Johnsen

Master's Thesis Autumn 2015

Interactive Zooming and Panning in Panoramic Video Using WebGL

Mattias Håheim Johnsen

Abstract

The Bagadus system is a largely automated tool designed to improve athlete performance by combining subsystems for sports analytics, player tracking and video capture, and is currently installed in several Norwegian soccer stadiums where it is used daily by coaches to observe and evaluate the performance of their athletes. It does this efficiently by automating the process of integrating these subsystems and their data. One important part of this system is the video capture subsystem which creates real-time panoramic videos from an array of HD cameras overlooking the field. The output of this image stitching pipeline is read by a panoramic video viewer application which takes the panoramic video and corrects for the image warping which results from the panorama projection process. This allows users to view the panoramic video through a virtual camera with features such as straight lines preserved. This camera is freely controllable by the user, who may zoom and pan the video in real-time.

The existing implementation of this viewer relies on the CUDA parallel computing platform, a closed proprietary technology requiring a discrete GPU from the same vendor. In this thesis we evaluate this viewer, and present a prototype of an alternative panoramic video viewer that uses open web standards such as WebGL. This prototype is capable of generating a virtual camera that interacts with high resolution panoramic video with equivalent graphical fidelity to that of the previous implementation on a much broader set of hardware and with far greater ease of use and deployment. We also introduce multiple improvements to the user interface and control methods, such as gamepad support, as well as techniques for increasing performance and reducing bandwidth consumption by segmenting and adapting the video stream based on how the virtual camera is observing the panoramic video.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgments

I would like to thank my supervisor Pål Halvorsen for his valuable guidance and feedback, Vamsidhar Reddy Gaddam for his technical support and valuable input on the previous panoramic view prototypes, and Ragnar Langseth for pitching the original idea that led to the writing of this thesis.

Additionally, I wish to thank my family and friends for their support and encouragement, especially my father Per Espen Johnsen, and my fiancee Christina Martinsson, without whom this thesis would not have been possible.

Oslo, August 3, 2015
Mattias Håheim Johnsen

Chapter 1

Introduction

1.1 Background

As sports clubs continue to seek out means of increasing their chances of victory, performance and game analysis has become an increasingly time and resource consuming necessity. Coaches and trainers are keen to quantify and better understand the performance of their athletes or teams, as well as that of their competition. By capturing and studying the relevant data from sporting events and training sessions they hope to study the strengths and weaknesses of their teams or individual athletes so that they may be improved upon. Likewise, these techniques have historically been used to discover exploitable weaknesses in the opposition, and to design effective counters to their strengths.

In the sport of soccer, analytics has a long history and has had such a large impact on the sport that books on the subject such as Soccernomics [2] have become popular among fans and are now becoming best-sellers. The field of sports has typically been an early adopter of technological innovation and today information systems play a big role in expediting the collection and analysis of data for sports analytics. Prozone [3], one such system, delivers its services to over 300 clubs and organizations worldwide and is indicative of the big market for these kinds of systems.

There exists a wide variety of products which have taken many different approaches to these systems. Prozone, for example, places emphasis on the gathering of player statistics such as heart rate, recovery, reaction times, acceleration, speed, and balance using sensors. By tracking players throughout the match one can correlate this data during post-match analysis and identify how players act when specific events happen in the game. Therefore, many of these systems also include either manual or automatic player tracking, and store this information for use with the other data. Prozone uses image analysis techniques to automatically track player positions during the game, while another approach demonstrated by ZXY Sports Tracking [4] tracks the players through sensor-equipped belts that they all wear during games. These belts communicate with radio transmitters placed along the field. The latter approach requires extra equipment to function, but will generally have higher accuracy in problematic situations where multiple players are clustered on the field, and obviously requires less intensive data-processing.

Other systems, such as Interplay [5], focus on video capture and annotation capabilities allowing a trained operator to mark relevant situations and events for later playback and analysis. The manual input required by such systems, as well as the effort needed to integrate the information gathered from the other types of systems we have discussed is significant. Coaches and trainers must either spend a lot of their time working with these tools themselves, train a dedicated operator to do it for them, or outsource the entire task to a company providing that service. All of these options are costly in some way or another, and as such there is a growing interest for automated systems that reduce the manual labor involved.

Bagadus is a project dedicated to creating one such system and aims to integrate all of the aforementioned functionality in as way that is as automated as possible. Bagadus is being developed as a joint effort between the iAD center for Research-based Innovation (project number 174867) and the **Department of Informatics at the University of Oslo (IFI)**. It is funded by the Norwegian Research Council, and is currently being put into use at multiple soccer stadiums in Norway. It combines real-time generation of a panoramic view of the soccer field where video captured by an array of multiple cameras is stitched together, with sensor readings gathered from the ZXY sensors system. Additionally

it integrates these systems with event annotations that are generated automatically or created by the coaches using a simple and efficient user interface for live or post-match annotation on their device of choice. This combination allows for quick and simple event identification and playback based on user queries, with no manual effort being needed to integrate the different types of data collected.

The stitched panoramic video generated by the Bagadus system provides end-users with an overview of the playing field. By using a panoramic video viewer application designed to display these videos without the warping created by the process, users may operate a virtual camera with which they can zoom and pan to focus on an arbitrary part of the action. Other parts of the Bagadus system allows for automatic event extraction, the creation of video summaries, and a host of other implemented and potential functionality. The novel part of the Bagadus approach is that the different subsystems that handle each task automatically integrate their data with each other, saving the end-user time and effort.

The panoramic video viewer is programmed to use a discrete GPU with support for the CUDA parallel computing platform in order to make the necessary computations that draws the virtual camera and unwarps the panoramic video. The CUDA platform is a closed proprietary technology and requires a powerful and expensive GPU from the vendor in order to run. This greatly limits the range of machines which may use the panoramic video viewer. Additionally, the current implementation is relatively difficult to install and maintain by laymen as it involves editing and compiling the source code on a UNIX operating system. Furthermore, the viewer has no **graphic user interface (GUI)** to speak of and must be run from the UNIX command line. This dependency on the CUDA API, compatible hardware and technical know-how is a major limitation.

1.2 Problem Definition

The main goal of this thesis is to improve the panoramic video viewer component of the Bagadus system. Multiple implementations have been made, one such being presented by Gaddam et al [6], but they all have a number of flaws related to performance, specific hardware or software requirements, and they are all difficult to install and use by people without extensive technical knowledge, which make up the bulk of the intended audience for the Bagadus system.

In this thesis we will explore the possibility of using currently available web technology in order to implement a panoramic video viewer that can run in a standard web browser as a web application. If possible, this should simplify the operation of the panoramic video viewer and make the software available on any common consumer device. We will first describe and evaluate the prior implementations of the panoramic video viewer and their design, in an effort to identify areas in which we may improve upon them. We will then describe the design and implementation of a fully-featured prototype using WebGL before comparing it and its performance to that of the previous iterations.

1.3 Limitations

Bagadus is a system currently undergoing active development on many fronts, but for this thesis we will assume that the state of the system remains as it was when work on the thesis began. Changes may very well have to be made before integrating the resulting prototype with the rest of the system, but that is outside the scope of this thesis.

The Bagadus Panorama Pipeline is currently also being improved, and while the deployment of the system at Ullevål stadium was underway when we began this thesis we will be working with a preexisting data-set from the Alfheim deployment.

Subjective evaluations of the video quality and user-friendliness of the applications described in this thesis have been performed exclusively by the authors and qualitative statements are therefore not the results of any significant user study.

1.4 Research Method

In this thesis we evaluate the design and implementation of preexisting panoramic video viewers developed for the Bagadus system prototype, as well as that of the WebGL-based panoramic video viewer developed as part of this thesis. This corresponds to the *design paradigm* put forth by the ACM Task Force on the Core of Computer Science [7], which can be summarized as iterating over four steps while constructing a solution to a given problem:

1. State requirements.
2. State specifications.
3. Design and implement the system.
4. Test the system.

1.5 Main Contributions

The main contribution of this thesis is the creation and evaluation of a new and improved panoramic video viewer prototype using the WebGL specification [8]. This web application allows for real-time zoom and pan in high quality panoramic video and corrects distortions caused by the panoramic video recording and stitching process. It exclusively uses open web standards to achieve results previously only possible using dedicated proprietary technologies, and runs so efficiently that it can display and interact with high resolution panoramic video in real-time on much of the current computer configurations in the wild. Our new prototype also includes a number of usability improvements over previous versions of the viewer such as additional user input methods, video controls, performance optimizations and techniques for reducing bandwidth constraints.

By implementing these viewers, we have shown that the WebGL technology is mature enough for use in sophisticated multimedia applications, and that this approach can provide an improved user experience for end-users of the Bagadus system. We have also demonstrated that it is possible to reach far greater target audiences with this new panoramic video viewer by virtue of it being a web application and thus accessible to everyone with a browser and an internet connection. These improvements open up a new path for further work on the Bagadus project.

1.6 Outline

In chapter 2 we will describe the Bagadus system and the state of it as work on this thesis began. A thorough understanding of the original implementation of the Bagadus system is helpful when considering the changes made to the panoramic video viewer application. In chapter 3 we will detail the theoretical background for the existing Bagadus panoramic video viewer and present how it was developed. Additionally we will evaluate how well each iteration of prototypes fulfill the requirements for the Bagadus project. In chapter 4 we will introduce the WebGL prototype and the improvements it brings to the Bagadus project. We will detail its design and implementation, discuss any alternative solutions we came across and argue for the choices we have made during the writing of this thesis. We will also compare the performance, user friendliness and system requirements of the new WebGL panoramic video viewer prototype with that of the previous implementations and explore any additional improvements that can be made. In chapter 5 we present and discuss additional improvements we made to the WebGL-based prototype after finishing and evaluating the basic functionality. In chapter 6 we will summarize the conclusions we came to and we will discuss any future work that may be of interest in regards to the project.

Chapter 2

The Bagadus System

In order to address demands for an integrated approach to soccer analytics the Bagadus system prototype was created [9]. This chapter describes the state of Bagadus and its subsystems. We will explain the basic idea behind Bagadus, and expand on the video capture and panoramic video stitching pipeline which is of primary interest to the contribution of this thesis. The tracking subsystem and the analytics subsystems and how these interact with the rest of the project will also be covered.

2.1 The Basic Idea

The interest for sports analytics is expected to continue to increase, but existing systems have generally been offline systems that require a large amount of manual work to integrate information from various data sources in order to create meaningful sport analytics. The Bagadus project is meant to fully integrate these systems in a way that allows not only for less work-intensive post-match analysis, but also for real-time presentation of significant events while the game is still being played. A prototype of the system has been developed using ZXY Sports Tracking sensors in cooperation with the Tromsø IL soccer club. A prototype is deployed at the Alfheim stadium in Tromsø, Norway [10].

Bagadus is made up of three different subsystems all working together for the main analysis application. The video capture subsystem consists of five high-resolution and shutter-synchronized cameras. These are arranged in such a way that they cover the field in addition to providing sufficient overlap between each other to identify common features and facilitate video stitching and camera calibration. This subsystem may be used to either switch between different video streams in order to follow players based on their sensor data or by parsing manual input commands. Additionally, it provides us with a panoramic view of the entire field made up of the individual streams stitched together. Image processing techniques are applied to make the stitching as artifact free and high quality as possible. The cameras are stationary, but a user may zoom, pan and even select and follow specific players with a virtual camera created from the panoramic view. An efficient video processing pipeline has been developed that is able to facilitate this in real time [11].

In order to identify, track and gather information on players, a tracking sensor subsystem is used where a lightweight belt with sensors is given to each player to wear. These sensors record the exact position of the player during the game, in addition to a range of biometric information. Tracking people using camera arrays is a much researched topic, and great strides have been made in regards to accuracy, but there are still errors, especially in real-time implementation. Capturing the exact position of players with sensors is a much simpler and foolproof approach and ZXY Sport Tracking provides the Bagadus system with this capability. Bagadus uses this to track either groups or single players in either of its camera modes.

Lastly, the analytics subsystem is an effort to allow coaches to move away from the traditional method of taking notes on paper while the game is being played or while reviewing hours of video footage. Instead Bagadus incorporates a system where the coach and his team can be equipped with a tablet or smartphone running an application designed to make registering predefined events or adding textual annotations easy. The design of this system prioritizes common actions so that annotations may be done as simple and efficient as possible, which is necessary as coaches want to be able to do this while

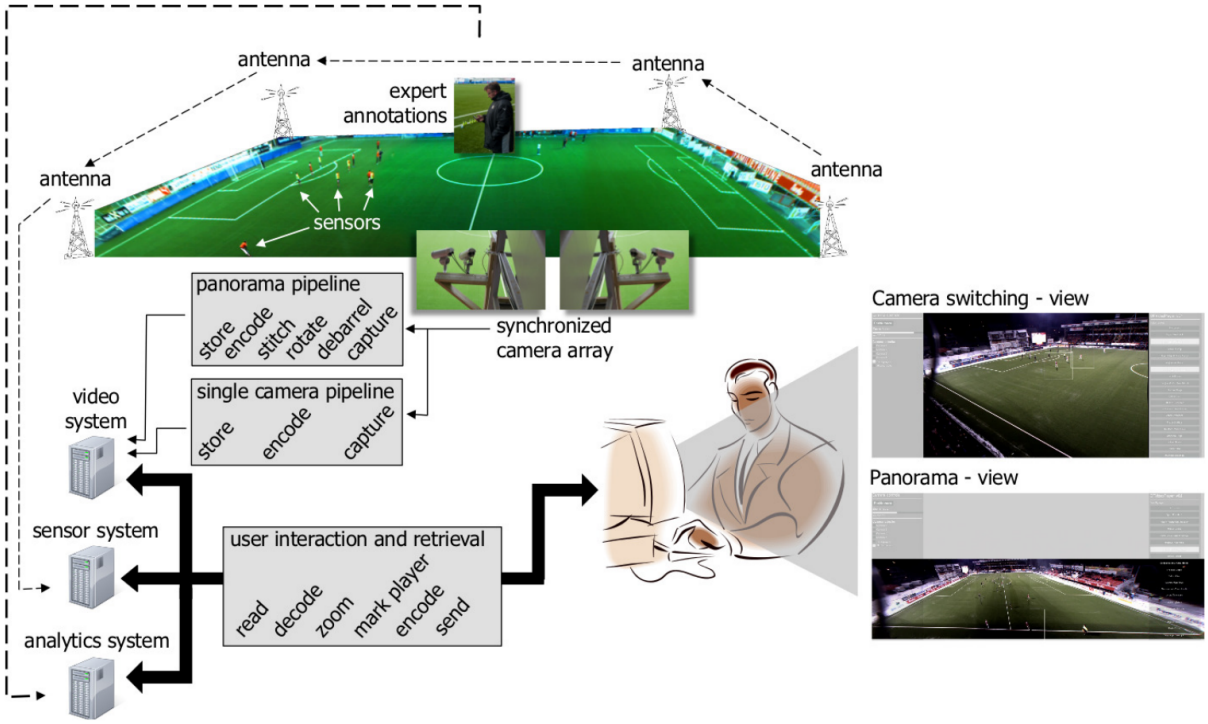


Figure 2.1: Bagadus set-up

watching the game live. The resulting semantic information is stored in the Bagadus analytics database and is automatically integrated with the captured video footage and relevant sensor data. This allows users to automatically extract the corresponding video footage of events. Of particular importance is the inclusion of the sensor data, which allows for automatic compiling and presentation of video events and summaries based on simple queries made to the system on the basis of that data.

This integration of spatial and temporal mapping of player locations and video with the sensor data allows for sophisticated video summaries being made in offline mode. A query could be made to the system asking for footage of all situations where a player had a heart rate under 150 while being within the penalty box of their own team and a video summary would be provided within seconds where creating such a video manually would take hours of work. Additionally, once that video summary has been presented the user is free to follow specific players, zoom in on areas of interest and read annotations made during the game. Online or real-time functionality is also included, allowing the user to rapidly focus on and follow specific players during the game or training session, or play back and review an event as soon as it has been annotated.

Trainers are not the only ones who might see the benefit of this work, as the pan and zoom functionality of the panoramic view could allow supporters and other end-users to utilize the panoramic video portion of the Bagadus system for entertainment purposes. The virtual camera that is generated from the panoramic view allows for a novel interaction with the game, giving fans the opportunity to be their own cameramen and follow the part of the action they most care about. Additionally, the semantic system developed for coaches is also possible to leverage for social networking purposes, allowing fans to create their own video summaries and extracting footage of events that they care about.

2.2 Video Subsystem

The video capture subsystem is the first step in the Bagadus work flow, and is one of the most important parts of the Bagadus system as it provides the image data used in all following steps. In this section we will explore the current camera set-up and how the image frames captured are synchronized so that they may be stitched together later in the process.

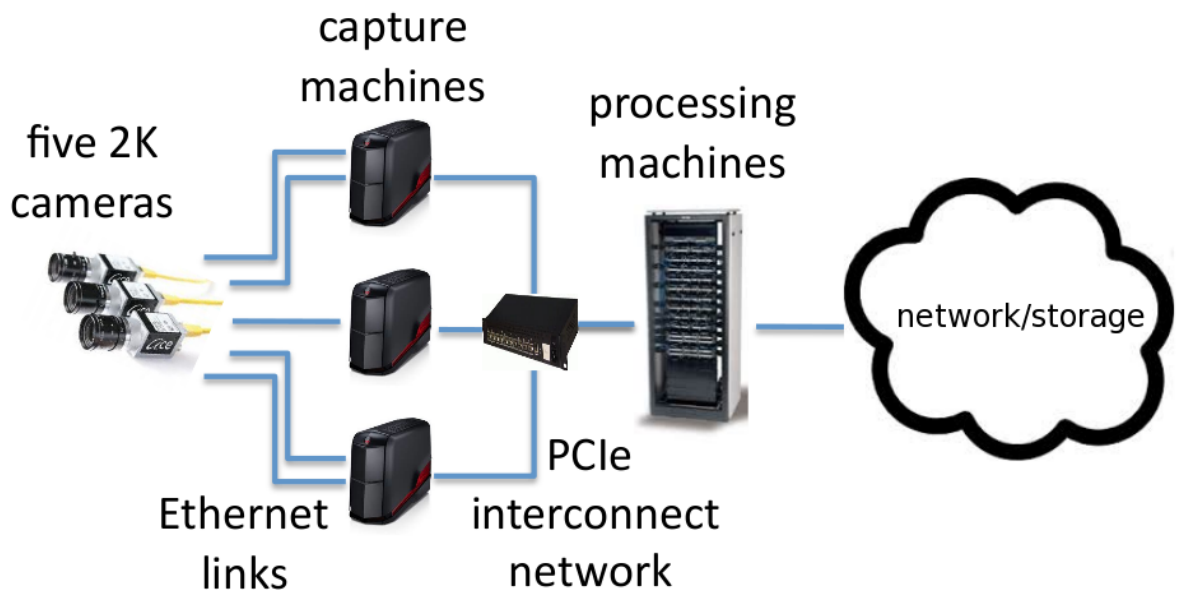


Figure 2.2: Bagadus video capture set-up

Figure 2.3: Bagadus camera array

2.2.1 Camera Set-up

The current camera set-up at Alfheim stadium consists of five Basler 2K High Resolution industry vision cameras, each of which delivers 50 frames of 2046x1086 video a second over a Gigabit Ethernet connection. Each camera is equipped with an 8mm lens that introduces no noticeable amount of image distortion. As the first prototype of this system used inferior lenses that required a lossy debarreling process to be run on each frame, this improvement has increased video quality and saves us a step of processing. Each camera is rotated 90 degrees to maximize the resolution of the resulting panorama and we end up with a vertical field of view of 66°.

The cameras are mounted in a circular pattern and adjusted so each camera is facing a common point 5 cm in front of their lenses, which is meant to reduce parallax effects [6]. Auto-exposure is handled by the center camera every fourth second and the results are transmitted by the camera reader module to the other cameras. As the lighting conditions in outdoor sport stadiums tend to vary dramatically this step is essential. For now, this appears to be the set-up that results in the best quality panoramic images while minimizing the amount of adjustments having to be made to each video stream before stitching.

2.2.2 Frame Synchronization

In order for the stitching process to be successful it is necessary for all five frames that will make up the panoramic image to be captured at the exact same time or artifacts will occur. To ensure this, an external device called a trigger-box has been custom built. This box sends a shutter pulse to all connected cameras

simultaneously and can be configured to do this at a variable rate. While the trigger box has only a set number of outputs, additional trigger-boxes may be daisy chained by attaching them to the first one. This will be used if we decide to extend the camera array with additional cameras in the future.

In the first prototype of the panoramic video pipeline two dedicated recording machines were used, with each machine handling two cameras for a total of four cameras. As this required further synchronization a custom made TimeCodeServer was installed so it could synchronize the internal clock for both machines.

2.2.3 Distributed Processing

In the iteration of the pipeline we are currently using the cameras require a lot more resources due to their increased resolution and the addition of a fifth camera. The demand for more bandwidth for each camera is especially high, and the capturing process has therefore been switched to a distributed one as shown on figure 2.2. Multiple machines are dedicated to processing the data, and since a single Gigabit Ethernet card cannot transfer this data quickly enough, we use Dolphin PCIe Interconnect cards to provide ultra-low latency Direct Memory Access between the video capture machines and the panorama processing machine. Each frame is transferred via a single DMA transfer and we achieve a rate of 19.41 Gbps, with a total latency of 1.26 ms per frame.

The cameras are managed using the Northlight library to handle frame synchronization, storage, encoding and the like. Northlight supports the Basler camera SDK, encodes using H.264 and perform color-space conversion using FFMPEG [12].

2.3 Analytics and Tracking Subsystems

As Bagadus is primarily meant to be a useful tool for coaches and managers it includes both a tracking subsystem and an analytics subsystem. The analytics subsystem is called Muithu and is an annotation system for registering events, while the tracking subsystem is provided by ZXY; a radio based positioning system with support for biometric sensors. In this section we will briefly describe both.

2.3.1 Muithu

The annotation system currently in use at Arnheim stadium is called Muithu. This system was developed in close coordination with the coaching team at Tromsø IL and allows coaches to annotate a game or training session using their smart-phone. One of the goals for this project was to make these tools as accessible and user-friendly as possible. Since coaches already carry cellular phones during practice this platform was seen as the natural fit for the Muithu application. Muithu was in use at Arnheim before the introduction of Bagadus and as such the integration of Muithu has always been a key point in the Bagadus analytics subsystem.

The user interface provided emphasizes tile-based user-interaction in order to make registering events as quick and intuitive as possible. These tiles can then be presented as a two-layer hierarchy with one layer representing event types such as training goal and the other representing individual players or all of them. Pressing a player tile will then show his associated event type tiles and dragging the player tile onto one of these will register that the corresponding event has happened for that player. During official games one might also have one layer of tiles represent defensive notations and one layer represent offensive notations. In this manner we aim to present a context sensitive user interface to make the use of this tool as simple or complex as is needed. An average of 3 seconds is required for the input of a single notation. These events are usually predefined, and can be customized by the coaching team. A match will usually include an average of 16 such events, as described by Johansen et al [13]. These events can then be combined with the rest of the sensor data gathered by Bagadus and can be retrieved and acted upon by the The Bagadus Player User Interface by way of preconfigured or user-specified SQL-style queries.

2.3.2 The ZXY System

The ZXY Sports Tracking system consists of sensor belts worn by players and 11 stationary radio receivers mounted around the stadium with overlapping field-of-view. The belt transmits information using the 2.45 GHz ISM band and each receiver is capable of calculating the position of the sensors independently. They can also receive biometric information from the included heart-rate sensor. The included accelerometer and compass polls position, speed and heading at 20 Hz, with additional less significant data such as total distance traveled being polled at the slower rate of 1 Hz.

These data are stored in an on-site SQL database and can be used by the Bagadus system to mark and follow one or more players live during the game or for more sophisticated queries by the analytics system. The positional data is also used to improve the accuracy and performance of the background subtraction algorithm used in the panoramic video generation pipeline. The ZXY Sports Tracking system was chosen as this was already in place at the first test site in Alfheim.

(a) Sensor belt used in the ZXY-system

(b) One of the ZXY-antennas at Alfheim stadium

Figure 2.4: ZXY sports tracking system

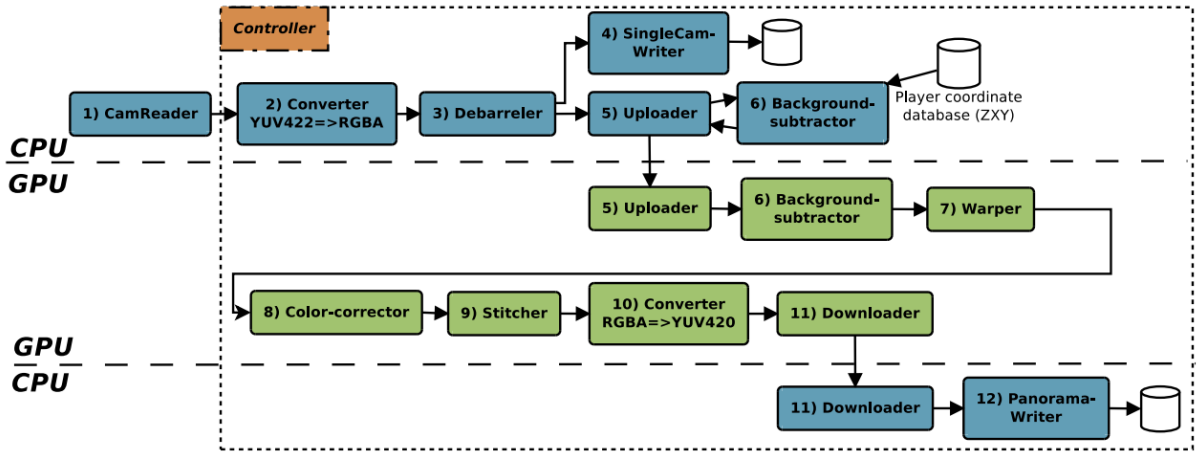


Figure 2.5: Distributed panoramic video stitching pipeline

2.4 The Bagadus Panoramic Video Stitching Pipeline

While Bagadus supports moving from one camera to the next and even using tracking analytics events to automatically move from one to another, we still want to be able to provide a complete overview of the field. Therefore, a panorama is generated by stitching the individual camera streams from each of our five cameras together into one seamless image. This is done through a multi-step process that are handled by modular sub-components of the panoramic video stitching pipeline as shown in figure 2.5. The modular design of this pipeline has the benefit of allowing individual modules to be improved or replaced if it becomes necessary.

A thorough explanation of each step in this process can be found in Tennoe et al [11] and will not be included in this thesis as we are mainly concerned with the end result of this process. To summarize, raw video frames are fetched from the cameras by the camera reader module, and the resulting streams are converted into a more manageable format and saved on disk. Meanwhile, the frames are being sent to a processing computer where they are frame synchronized once again. Once we have a complete set of frames from each camera these sets are uploaded to the GPU and handled by the panoramic image stitching module where cylindrical projection is used to stitch the frames together. We use the NVIDIA CUDA architecture to do this in parallel, as it is essential for this step to be able to perform in real time [14]. The current system manages to do this at a full 50 frames per second. These frames of panoramic video are encoded by the video encoder module into H.264 and may be delivered directly to clients or stored.

The resulting panoramic frames will of course feature distortions typical of cylindrical panoramas as shown in figure 2.6. These are undesirable and we therefore require a viewer able to compensate for these distortions and capable of presenting an image where geometric details such as straight lines remain the way one would expect.

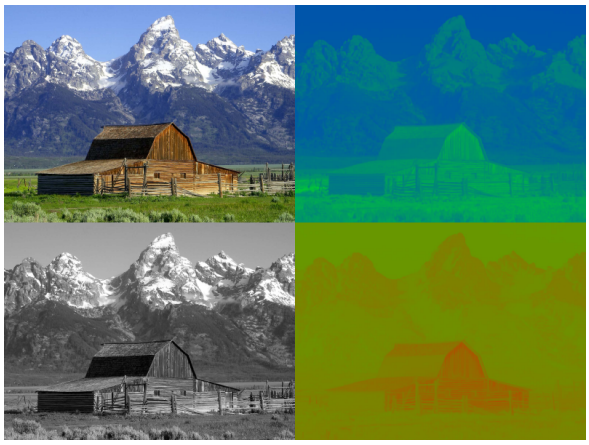
2.4.1 Color Formats

The Bagadus Panorama Pipeline and the applications handling its output use the YUV420 color format, which has some marked differences from the more common RGB format. We will explain them in short here because they are relevant to how the viewer has been implemented. For a more thorough explanation see *Bagadus: Next generation sport analysis and multimedia platform using camera array and sensor networks* by Simen Sægrov [1].

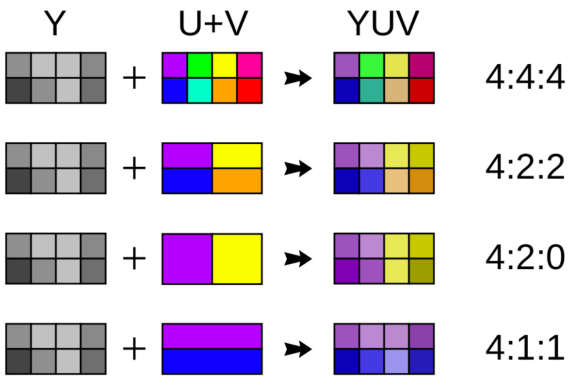
The RGB or RGB24 format uses an additive color model in which values for the colors red, green and blue are encoded with eight bits for each component and blended to represent the color of a single pixel. This provides 16,777,216 different combinations of R, G, and B values, which translates into millions of hue, saturation and lightness shades. Adding an alpha channel for the translucency of a pixel as a fourth 8 bit component gives you the format known as RGBA or RGB32.

According to van der Branden Lambecht [15], the YUV format is encoded in a way that takes human

Figure 2.6: Generated cylindrical panorama



(a) In clockwise order: Original image, U and V chrominance components, Y luma component



(b) YUV chroma sub-sampling schemes

Figure 2.7: YUV color model examples [1]

perception into account and allows for reduced bandwidth of chrominance components (chroma sub-sampling) seeing as the human eye is more sensitive to gray scale information/luminance than it is to color information like position and motion. The YUV model defines color as being made up of one luma-component called Y which is analogous to brightness, and two chrominance (color) components, U and V.

This format originated in the move from black and white television systems to color television where color components were added to the preexisting Y (luma/brightness) component. The exact ratio of sub-sampling is usually denoted in the format description. For example YUV422 halves the chroma information as compared to RGB (4:4:4), reducing the bandwidth required by a third. See figure 2.7(a) for a visual example of the three YUV channels and what each component brings to the image. See figure 2.7(b) for an illustration of the different chroma sub-sampling schemes.

2.4.2 Cylindrical Projection

The process of mapping the panorama to the cylinder can be visualized as a process where we have the camera array sitting in the middle of a virtual cylinder where every image captured by the cameras is considered a cropped plane positioned tangentially to the virtual cylinder and orthogonal to the axis of the camera view. These images are painted onto the cylinder by computing every pixel as an (interpolated) value of a ray emanating from the center of the camera to where it intersects with the image plane. [6]

The radius (r) of the virtual cylinder is defined by the width (Ws) of the source images and the **field of view (FOV)** of the camera:

$$r = \frac{Ws}{2 * tan(\frac{fov}{2})} \tag{2.1}$$

The axes of the camera view form a plane (L) which is orthogonal to the rotation axis of the cylinder. The angle of the camera view axes between two neighboring cameras in the camera array is approximately 28.3 degrees when the center camera is considered the 0 degree angle. The angle of a camera corresponds to the angle of the source image it captures and we define this angle as (α). Once unrolled, the virtual cylinder forms a Cartesian coordinate system where (0,0) is the intersection of the viewing angle of the center camera with the cylinder and where the X axis is the intersection of L and the cylinder. Every pixel coordinate (Tx, Ty) on the unrolled cylinder determines the corresponding horizontal (θ) and vertical (ϕ) angles of a ray that emanates from the camera center and passes through this coordinate.

$$\theta = \frac{Tx}{r} \qquad\qquad \phi = arctan(\frac{Ty}{r}) \tag{2.2}$$

We want to map every pixel from the source image to a pixel in the unrolled cylinder (Tx, Ty), so first we select the source image with a value of α that is closest to θ and subtract α, thereby centering the coordinate system on the viewing axis of that particular camera. The point in 3D space where the ray intersects with the image plane (x', y', z') is then determined by:

$$z' = r \qquad\qquad x' = tan(\theta) * z' \qquad\qquad y' = tan(\phi * \sqrt{z'^2 + x'^2}) \tag{2.3}$$

This relationship can be seen in figure 2.8. It is important to note that this algorithm requires perfectly aligned and rotated cameras, and while the camera mount shown in figure 2.3 is quite precise there are still small adjustments that must be made to achieve the correct alignment. A small vertical correction can be applied per camera with no additional complexity.

For every frame we have to create a standard rotational matrix that compensates for rotation around any of the three axes of the camera. The Cartesian coordinates of the pixels on the cylinder are multiplied with this matrix, which in turn rotates the cylinder to the correct position before we project the images onto it. The horizontal and vertical angles θ and ϕ are then found based on these new coordinates, but this process is unfortunately significantly more costly than the one shown earlier:

$$\theta = arctan(\frac{x}{z}) \qquad\qquad \phi = arctan(\frac{y * sin(\theta)}{x}) \tag{2.4}$$

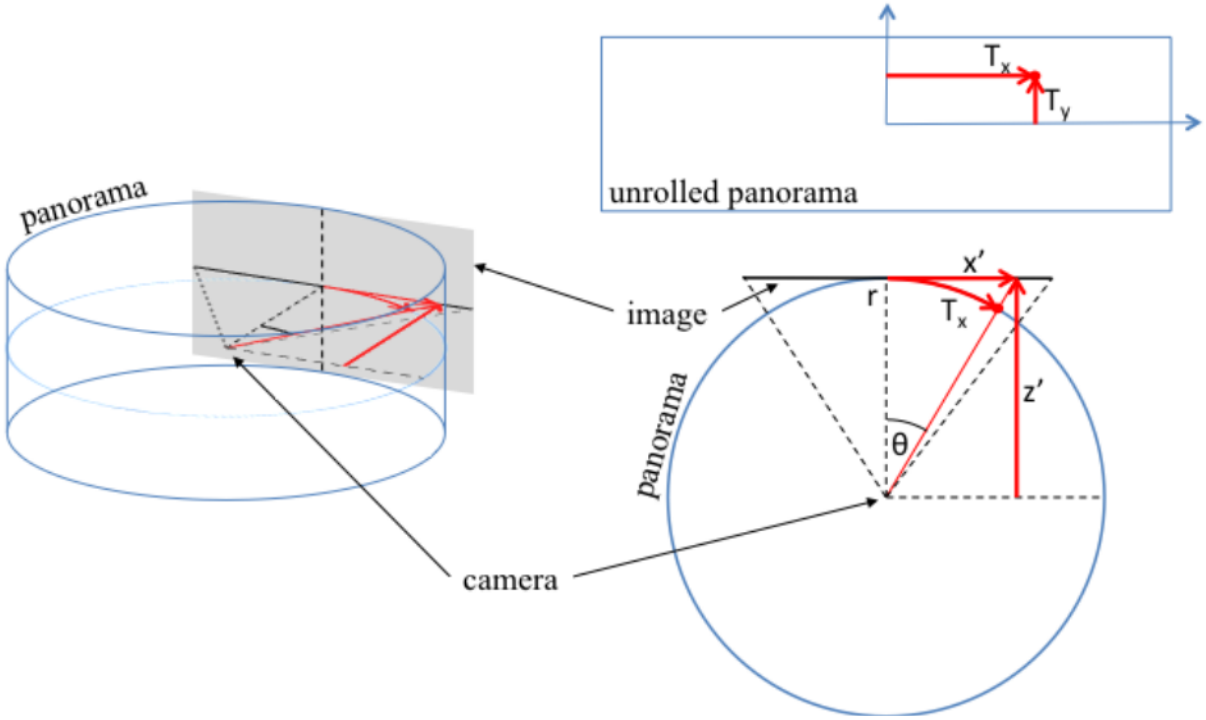


Figure 2.8: Mapping pixels from a captured image to a panorama

These calculations involve some complicated operations such as an inverse tangent and a square root for the calculation of every pixel, however since this is done in a linear fashion it is well suited for parallelization and as we shall see in the next chapter the implementation of this solution was later ported to the CUDA parallel computing platform in order to make use of the superior parallelization capabilities that GPUs have.

2.5 Summary

In this chapter we have described the Bagadus system prototype and its panoramic video pipeline. We began by looking at the basic idea behind the project and its intended goals before detailing the hardware components that make up the video capture portion of the system. We then looked at the individual sub-components responsible for each part of the Bagadus functionality. Technical details about the sensor data integration or the stitching and encoding process in the panoramic video pipeline are not relevant to this thesis, but the mapping of the cylindrical panorama is. Therefore we have gone through that algorithm step-by-step. This part is of significance as it will be the the one used when displaying the panorama using the panoramic video viewer implementations detailed in chapter 3, as well as serving as the theoretical background for our simplified idea of doing this kind of unwarping as detailed in chapter 4.

Chapter 3

The Bagadus Panoramic Video Viewer

3.1 Motivation

In this chapter we will examine and evaluate the existing panoramic video viewer implemented by Gaddam et al [6] for the Bagadus system so that we may understand the theory behind it, the requirements and specifications for a panoramic video viewer, and identify which areas we may improve upon when we implement our own prototype.

The Bagadus panoramic video viewer presents an end-to-end real-time system for interactively zooming and panning into high-resolution panoramic videos. Instead of zooming by merely cropping into a perspective panoramic video, our idea is to use a cylindrical panorama as an intermediate representation and then generate an arbitrary virtual camera view emanating from the position of the camera array. The cylindrical panoramic video is warped as a result of the projection process and these distortions are particularly obvious when looking at what should be straight lines in the world that we filmed. The virtual camera allows the user to freely zoom and pan the camera while the program corrects the perspective of the displayed video to one without the aforementioned distortions. The resulting experience for the user is much like having a real camera of their own to control, all in real-time.

Results from previous experiments indicate that these virtual views may be generated below the frame-rate threshold with one unoptimized GPU implementation being able to generate a virtual view frame in about 10 ms. This indicates that future implementations should be able to bring this time down far enough that even middle of the road consumer devices are able to de-warp the panoramic video in real time.

In addition to being a tool for coaches we hope to explore the possibility of making this functionality available to fans and spectators both inside the stadium and in their own homes via internet streaming. The overview afforded by the panoramic view and the flexibility and ease of use provided by the viewer would allow coaches to easily tailor their user experience when reviewing matches, training sessions, and footage of retrieved Muithu events. The same functionality could be used to allow fans greater control of their viewing experience by acting as their own cameraman, as shown in by Gaddam et al [16], and possibly even share this experience with others through social networks in the future.

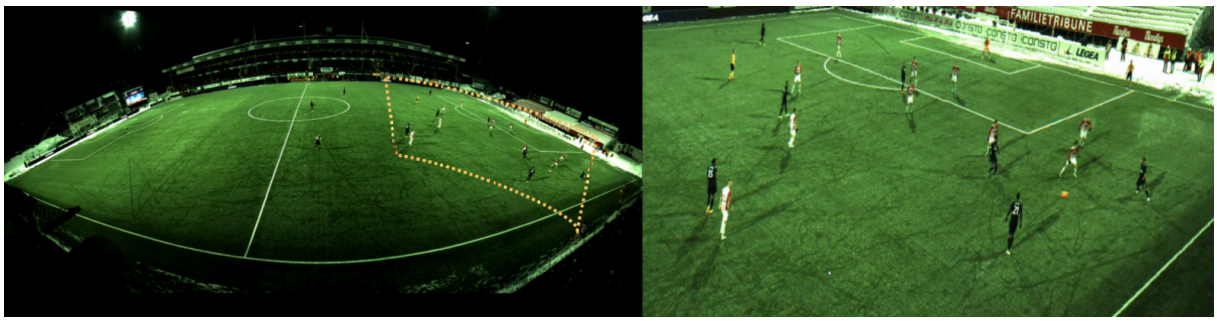


Figure 3.1: Panoramic video with labeled region of interest (left) and the generated virtual camera of the region (right)

In order to manage all of this the panoramic video viewer must be able to fetch and decode the panoramic video, create a virtual camera and handle user input. The video data should ideally be streamed and we are currently using HTTP segment streaming in order to serve the program with video data. Alternatives such as adaptive HTTP streaming are being explored. Test data is currently hosted on an Apache web server as a number of 3 second video clips containing exactly $3 \times$ **frames per second (FPS)** frames along with a manifest file [17]. This file is read by the client periodically and lets the client know when the next file is ready to be downloaded. The video files that are downloaded are kept cached and ready to be processed. The bandwidth required is substantial because of the large resolution of the files, and H.264 compression is used to help mitigate the problem.

3.2 Related Work

Panorama images are popular for both research and artistic purposes. The generation of panoramic images is a well researched field and a number of image stitching systems have been developed to create them (see [18–22]). Manual and semi-manual panoramic image stitching functionality are also common features in mobile phones and digital cameras.

The appeal of panoramic images is not limited to merely stitching images into a larger whole. Users want to interact with the images, and be able to view them without the warping effects commonly found in panoramic images. As a result panoramic image viewer applications are also quite common, and can vary from specialized research tools allowing researchers at NASA to zoom and pan in high-resolution images captured by robots on the moon as described by Fong et al [23], to consumer software for panning in home photography. The book *Interactive panoramas: techniques for digital panoramic photography* [24] lists and compares a selection of software packages that allows for user interaction with panoramic images.

Panoramic videos present us with additional challenges, especially if you want to process the images quickly enough to present them in real time, but generally build on the techniques mentioned above. A number of papers describe projects where panoramic video is processed in such a way as to allow users to create arbitrary video streams or virtual perspectives, with the end goal of allowing users to interactively zoom and pan as they please [25–27]. This functionality is also the main idea for the panoramic video viewer prototypes we will be presenting in this chapter. There exists published papers [1, 16] describing the resulting application and how it has been integrated into the Bagadus system so we will focus mainly on the basics of their implementation, how it evolved from prototype to prototype and how each of them performs, as we will be building on this progress and comparing their results to the WebGL-based prototype we will present in following chapter.

3.3 Architecture

The panoramic video viewer prototypes all take the location of one or more video files as input, and reads them frame by frame with a simple class that uses functions from the FFMPEG library. The main loop of the viewer takes each frame read by the videoReader class and has them processed by a processing kernel. The kernel makes use of a rotational matrix defined in part by user input, which is also checked for each time the loop runs. The viewer thus continues until the video ends or it receives user input telling it to halt and exit.

In the rest of this section we will describe and discuss the main technologies that are used by the various panoramic video viewer prototypes that we will look at in this chapter.

3.3.1 OpenCV

Open Source Computer Vision (OpenCV) [28] is a cross-platform open source library of programming functions mainly intended for real-time computer vision, and as such contains methods for acquiring, processing, analyzing, and understanding images and other high-dimensional data from the real world. While these are used mainly in an effort to produce symbolic or numerical data that computers can act on, the OpenCV library also includes a wide variety of image processing functions that has made it popular

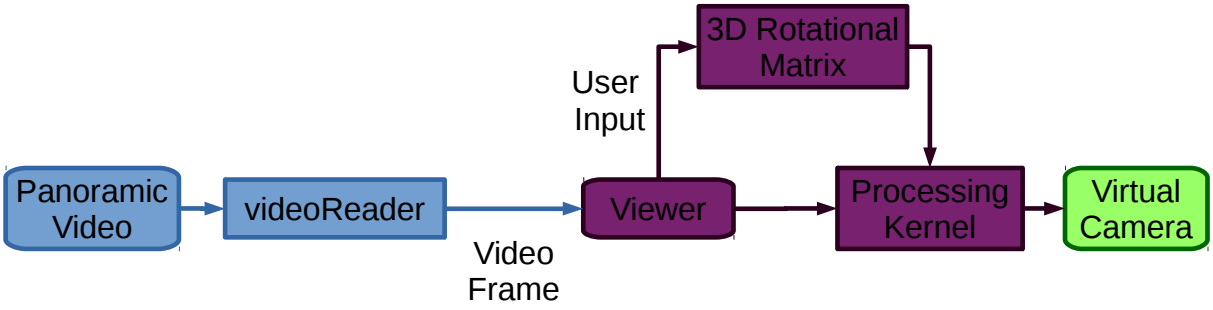


Figure 3.2: Panoramic video viewer prototype architecture

as a general purpose library for handling images. It is functions are primarily intended to run on the CPU, but CUDA and OpenCL-based interfaces now allows it to take advantage of GPU optimizations as well, although this functionality is not used to optimize this implementation.

OpenCV includes a primitive image data type called *cv::Mat* that is used for all OpenCV operations in the Bagadus system. Frames can be converted from this format and saved to disk as JPEGs at any time which makes it convenient when debugging. This format is also used to build the virtual view in the CPU implementation of the panoramic video viewer. OpenCV is also used by the first implementation to generate a window and show the resulting image inside it, as well as facilitate updates when the keyboard is interacted with in order to pan or zoom with the virtual camera.

3.3.2 OpenGL

The Open Graphics Library (OpenGL) is a cross-platform **application programming interface (API)** used for drawing 2D and 3D graphics. It is mainly used in conjunction with a GPU to achieve hardware-accelerated rendering, although software implementation of the API remains a possibility. The API consists of a number of functions usable by the client program. These are written in a language similar to C, but are language-independent. As a result, there exists a multitude of language bindings for OpenGL, which combined with the open nature of the API and its lack of platform-dependency makes it one of the most universally adopted graphics API in the computer industry. The popularity of **general purpose GPU (GPGPU)** libraries such as OpenCL and CUDA has opened up the possibility of using the GPU for data-processing without having to work within graphic-oriented frameworks such as OpenGL. However, since GPGPU libraries focus their efforts on computation, rather than graphics, they are not viable replacements. Instead, we can in those cases where we are not only looking to speed up calculation, but also display our results as quickly as possible, have CUDA and OpenGL work together: sharing memory buffers and efficiently communicating data between CUDA and the graphics pipeline.

The prototypes described in this chapter all use OpenGL in order to display graphics using the GPU. We will also be using two helper libraries: the **GL Utility Toolkit (GLUT)**, and the **GL Extensions Wrangler (GLEW)**. The former provides a cross-platform interface between the windowing system and OpenGL and will be used for for handling windows and input in the GPU-based implementations. The latter helps load and work with different version of OpenGL and extensions for OpenGL, with GLUT being our primary concern. Most implementations of OpenGL follow a set order in which various processes are performed so graphics might be rendered; this is called the OpenGL graphics pipeline. First, memory buffers managed by OpenGL are filled with vertex arrays by the host program. Vertex arrays are arrays of vertices; data structures describing the attributes (primarily the position) of a point in 2D/3D space. These points again describe flat surfaces, usually triangles, which serve as the building blocks for graphic objects. Each vertex not only contains the location of a corner of a surface, but also additional information needed when rendering the object, such as color or texture information.

The vertex arrays are then projected into the scene. This process involves assembling the vertices into triangles, and transforming the abstract attributes of the object model (such as spatial coordinates) into the coordinate system of our scene. The vertices are transformed by a projection matrix that defines the perspective in our scene, and any geometry that fall outside of a set of user-defined clipping planes is removed. The remaining primitives are mapped to the view-port or screen space, and are then rasterized

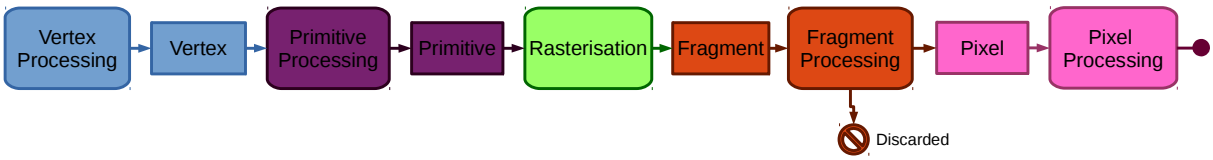


Figure 3.3: OpenGL ES 2.0 pipeline overview

into fragments the size of a pixel. These fragments are processed by a shader which commonly handles color, texture mapping and lighting, and are drawn to the frame-buffer, which displays the resulting 2D image on our screen. Fragments who fail any number of tests during this process, such as containing information that would cause it to try writing to a pixel of the frame buffer owned by another process, or those who fall outside of a designated rectangular area of the frame buffer are discarded at this point.

An interesting feature of OpenGLs **frame buffer objects (FBO)** is that they may be used for a variety of clever effects. For example we can take these images, which at this stage would normally be drawn on the monitor, and render them to a texture instead using a pixel shader.

3.3.3 CUDA

Using GPUs for general purpose computational tasks has become popular with the widespread availability of such devices. Processing computer graphics usually entail a lot of linear algebra and matrix operations which require a high degree of parallelization to run efficiently. The hardware architectures designed to support this were discovered to fit a much wider range of general computing tasks and the computer science field quickly started leveraging this technology in their research. These days even consumer graphic cards include APIs that simplify this process. GPU manufacturers provide their own programming environments especially suited for this use and are even creating new products based on the GPU architecture (like the NVIDIA Tesla) that eschews the ability to output images in favor of being purely general purpose computational devices.

CUDA is one such programming environment and was created by NVIDIA for use with their GPUs. It provides programmers with easy access to the memory of the parallel computational elements in the GPU and their virtual instruction set [29]. This allows programmers to use the CUDA-enabled GPU as if it was a CPU with the important difference that a GPU focuses on executing a great many simultaneous threads rather than just a few. Data is copied from main memory onto that of the GPU, the CPU provides the instructions for how it is to be processed and this is executed in parallel on the many cores of the GPU before the results are copied back to the main memory. This processing flow is shown in figure 3.4.

CUDA C extends C by letting a developer create C functions that are called kernels, which when called are executed a certain number of times in parallel by a certain number of CUDA threads, as opposed to only once like most regular C functions. Each thread has a unique id, which is given as a 3-component vector. This way, threads can be grouped and identified in anything from a one-dimensional to a three-dimensional way. We call these groups of threads blocks, and arranging the threads in such a fashion is a natural fit when computing across individual elements in domains that resemble vectors, matrices or volumes. Each thread executes the given kernel and will generally make use of its index to process parts of the data-set in a manner which means that the total number of threads will cooperatively deal with the entire data-set. Multiple blocks may be created and arranged in a 2D array we call a grid.

CUDA functionality can be used through the native CUDA C/C++ and CUDA FORTRAN programming language extensions or via a myriad of CUDA-accelerated libraries and compiler directives (OpenAAC being the most common). The CUDA platform also provides additional functionality in the form of a variety of GPU-accelerated Libraries. Among these is the **NVIDIA Performance Primitives library (NPP)**, which is a collection of GPU-accelerated image, video, and signal processing functions that NVIDIA promises will deliver 5-10x the performance of similar CPU-only implementations. Third party wrappers for most of this functionality now exist for a wide range of programming languages and computational software suites.

An alternative to CUDA is OpenCL by the Khronos Group. This platform can take advantage of many of the same capabilities that CUDA does, but is considered less platform dependent [30] as it is

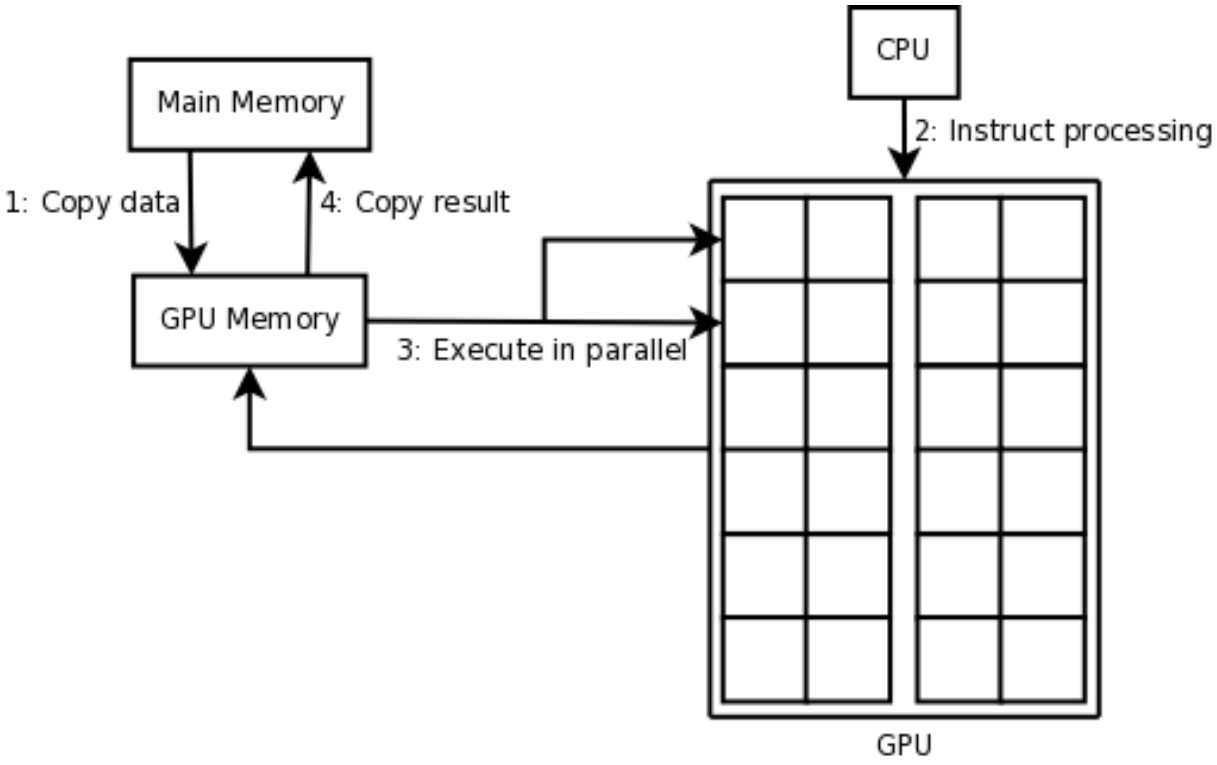


Figure 3.4: CUDA processing flow

an open standard which is supported by a wide variety of GPUs (and CPUs). Achieving performance on par with CUDA is possible [31], but requires tweaking for each platform it is to be run on. The cost for being multi-platform is therefore not negligible and as a result CUDA has been chosen as the primary parallel computing platform for the Bagadus system and associated applications.

3.4 CPU Prototype

The first implementation considered for a panoramic video viewer was one that simply looped through every pixel in the virtual view looking for the positions where the rays intersect with the panoramic plane. It uses the OpenCV and OpenGL libraries to carry out the computations and show an interactive panoramic video where many of the artifacts of the cylindrical projection have been compensated for.

3.4.1 Implementation

The CPU implementation is written in C++ and compiles to a single executable. It reads frames from a H.264 encoded video file and stores them as a texture, it then parses input from the keyboard, creates a virtual camera, gathers the necessary image data and displays the image on the screen.

A virtual view is initialized using OpenCV and the program enters a loop where it reads a frame from the video file and processes it while also waiting for user input via the keyboard. These input commands instructs the program to either quit or to move the virtual camera. To reduce the amount of calculations that has to be done every time we want to fetch a pixel through the virtual camera we build a 3D rotational matrix called inverse matrix M from three matrices K, Rx and Ry which describe the camera orientation.

$$\left[\begin{bmatrix} -focal & 0 & width/2 \\ 0 & -scope * focal & height/2 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(x) & sin(x) \\ 0 & -sin(x) & cos(y) \end{bmatrix} * \begin{bmatrix} cos(y) & 0 & sin(y) \\ 0 & 1 & 0 \\ -sin(y) & 0 & cos(y) \end{bmatrix}\right]^{-1}$$

Table 3.1: Creation of inverse matrix M

Table 3.1 shows the matrix in question, which translates into the code in listing 3.1. This code must be recalculated every time the virtual camera is interacted with but is otherwise constant for every frame.

```
theta_z = -0.4 * theta_y;

//First row
M[0][0] = (-cos(theta_y) * cos(theta_z) / focal);
M[0][1] = ((cos(theta_z) * sin(theta_x) * sin(theta_y) - cos(theta_x) *
 ↪  sin(theta_z)) / (focal*scale));
M[0][2] = (1/(2 * focal * scale)) * (VIRTUAL_WIDTH * scale * cos(theta_y) *
 ↪  cos(theta_z) + cos(theta_x) * (-2 * focal * scale * cos(theta_z) * sin(theta_y)
 ↪  + VIRTUAL_HEIGHT * sin(theta_z)) - sin(theta_x) * (VIRTUAL_HEIGHT *
 ↪  cos(theta_z) * sin(theta_y) + 2 * focal * scale * sin(theta_z)));

//Second row
M[1][0] = (cos(theta_y) * sin(theta_z) / focal);
M[1][1] = (-cos(theta_x) * cos(theta_z) - sin(theta_x) * sin(theta_y) *
 ↪  sin(theta_z)) / (focal * scale);
M[1][2] = (1/(2 * focal * scale)) * (VIRTUAL_HEIGHT * cos(theta_x) * cos(theta_z) -
 ↪  2 * focal * scale * cos(theta_z) * sin(theta_x) - scale * VIRTUAL_WIDTH *
 ↪  cos(theta_y) * sin(theta_z) + 2 * focal * scale * cos(theta_x) * sin(theta_y) *
 ↪  sin(theta_z) + VIRTUAL_HEIGHT * sin(theta_x) * sin(theta_y) * sin(theta_z));

//Third row
M[2][0] = -sin(theta_y) / focal;
M[2][1] = -cos(theta_y) * sin(theta_x) / (focal * scale);
M[2][2] = (1 / (2 * focal * scale)) * (2 * focal * scale * cos(theta_x) *
 ↪  cos(theta_y) + VIRTUAL_HEIGHT * cos(theta_y) * sin(theta_x) + scale *
 ↪  VIRTUAL_WIDTH * sin(theta_y));
```

Listing 3.1: The creation of inverse matrix M

The frame processing part runs through every pixel of the virtual view and fetches the corresponding pixel from the frame texture. Since we are operating with floating point values nearest-neighbor interpolation is used to solve the problem of what value to give to a pixel when the ray does not intersect with a specific given pixel in the texture, but rather some non-given floating-point position. It simply selects the value of the closest given pixel. The processing code snippet is shown in table 3.2.

As you can see, these calculations are quite expensive, including the calculation of an inverse tangent and a square root for every pixel. However, these calculations are well suited for parallelization and a second implementation was written to utilize the parallelization capabilities of a GPU in order to increase performance. It was decided that the NVIDIA CUDA library would be used for this purpose, as they are the market-leader in this field and compatible hardware is readily available.

3.4.2 User Experience

This implementation of the virtual viewer primarily runs on the Linux operating system, but should be able to run on any UNIX-based system with little effort. It requires the OpenCL and FFMPEG libraries as well as the OpenGL library and a graphics card that supports it. Users must compile the program themselves, and run it from the command-line. Since the program is a prototype it will only display a locally stored panoramic video and the location and name of this video is hard-coded into the program. This means that if we want to change which video to display we must open and edit the source code, or alternatively overwrite the existing test file. There is also no support for streaming stored or live videos.

The program displays the panoramic video in a window and allows the user to pan, tilt and zoom the camera using simple keyboard controls that are explained in a read-me file that comes with the program. The frame rate is extremely poor, well under 1 frame per second even with the video scaled down to a 1080p format. Keyboard controls are likewise unresponsive and cumbersome to use as a result of the poor performance. As can be seen in figure 3.5 there are no video controls or GUI elements that give any indication on where we are in the video or any other type of information. As shown in figure 3.6 the

```
for (int j = 0; j < VIRTUAL_HEIGHT; j++) {
  for (int i = 0; i < VIRTUAL_WIDTH; i++) {
    s1 = M[0][0] * i + M[0][1] * j + M[0][2];
    s2 = M[1][0] * i + M[1][1] * j + M[1][2];
    s3 = M[2][0] * i + M[2][1] * j + M[2][2];
    tx = std::atan2(-s1, s3) * ((double)panoWidth) / panoFOV + ((double)panoWidth) /
 ↪  2;
    ty = ((double)panoHeight) / 2 - ((double)panoHeight) * s2 /
 ↪  std::sqrt(s1*s1+s3*s3);

    //Fetching the texture
    if (tx >= 0 && tx < panoWidth && ty >= 0 && ty < panoHeight) {
      yp = inputY[(panoWidth*(int)floor(ty)+(int)floor(tx))];
      up = inputU[(panoWidth*(int)floor(ty/4)+(int)floor(tx/2))];
      vp = inputV[(panoWidth*(int)floor(ty/4)+(int)floor(tx/2))];

      tmpb = yp + 1.772*(up-128);
      tmpg = (yp - 0.34414*(up-128) - 0.71414*(vp-128));
      tmpr = yp + 1.402*(vp-128);

      virtualView.at<cv::Vec3b>(j,i)[0] = (unsigned char) (tmpb < 0 ? 0 :(tmpb >255
 ↪  ? 255 : tmpb));
      virtualView.at<cv::Vec3b>(j,i)[1] = (unsigned char) (tmpg < 0 ? 0 :(tmpg >255
 ↪  ? 255 : tmpg));
      virtualView.at<cv::Vec3b>(j,i)[2] = (unsigned char) (tmpr < 0 ? 0 :(tmpr >255
 ↪  ? 255 : tmpr));
    } else { virtualView.at<cv::Vec3b>(j,i) = defPixel; }
  }
}
```

Listing 3.2: Video frame processing in the CPU-based panoramic video viewer

video quality is not great, due to the down-scaling needed to make the video play and the lack of proper interpolation.

Figure 3.5: CPU-based panoramic video viewer GUI example

Figure 3.6: CPU-based panoramic video viewer image quality in detail

3.5 GPU1: First GPU Prototype

In order to increase performance the prototype was ported to the GPU, using the CUDA library as previously mentioned. The main change in this new prototype is that the calculation of the ray intersection with the panoramic plane and subsequent fetching of the corresponding pixel are both performed on the GPU. Video decoding is still performed on the CPU and each frame is then transferred to the GPU for processing. The virtual camera image is built on the GPU and then transferred back to the host (CPU) for displaying (or alternatively storage).

3.5.1 Implementation

This implementation is very similar to the CPU implementation. It is mainly written in C++ and likewise compiles to a single executable, but the processing code from the CPU implementation shown in table 3.2 is ported to CUDA C as a kernel that can then be executed in parallel. This should speed up drawing of the virtual view significantly. A video reader class using FFMPEG libraries handle the reading of videos and passes frames to the virtual viewer class. Like before, the video reader is stated to support a variety of formats, but expects the YUV420p color space by default. As with the CPU viewer this video reader expects to read a local pre-rendered video file with a hard-coded location.

In addition to allocating memory for the 3D rotational matrix, and initializing the virtual view with OpenCV this implementation also allocates memory for a corresponding M matrix on the GPU and configures the device with identical width, depth and FOV as the CPU-side virtual view. We now have a set-up very similar to that in the CPU implementation, but with the addition of a corresponding GPU environment to run the calculations on. The program then enters the drawing loop, where the matrix is updated (on the CPU) and sent to the GPU together with the video frame. The function makeView is then called, as shown in listing 3.3 which in turn initiates the execution of the processing kernel on the GPU.

The kernel divides the image frame to be processed into 16x16 pixel blocks of 16x16 threads, meaning every pixel value is computed by an individual thread. Each thread processes their pixel in an identical way to that described in the CPU implementation, as can be seen in listing 3.4 The cudaThreadSynchronize call will block execution until every thread has finished its work. The result is then copied back to the memory set aside on the CPU and displayed through the use of the OpenCV library.

```
void makeView(int iw, int ih, int panoWidth, int panoHeight, double panoFOV,
  ↪  unsigned char *source, unsigned char *sourceU, unsigned char *sourceV, unsigned
  ↪  char *dest, float* M){

  dim3 blocks(iw/16, ih/16);
  dim3 threads(16, 16);

  // Execute the kernel
  makeView_kernel<<<blocks,threads>>>(iw, ih, panoWidth, panoHeight, panoFOV,
  ↪  source, sourceU, sourceV, dest, M);
  cudaThreadSynchronize();
}
```

Listing 3.3: Launching of video frame processing kernel in the first GPU panoramic video viewer

3.5.2 User Experience

As with the first prototype, the implementation of the second prototype is primarily built for UNIX operating systems and requires all the aforementioned libraries in addition to the CUDA library and proprietary NVIDIA drivers. As the program is only distributed as source code significant effort must be made to fulfill these dependencies and build the executable. Our development machine (detailed in section A.1.1) was initially unable to compile and run the first GPU prototype as the GPU that was

```
__global__ void makeView_kernel(int iw, int ih, int panoWidth, int panoHeight,
 ↪  double panoFOV, unsigned char *source, unsigned char *sourceU, unsigned char
 ↪  *sourceV, unsigned char *dest, float *M){

  int x = (blockIdx.x * blockDim.x) + threadIdx.x;
  int y = (blockIdx.y * blockDim.y) + threadIdx.y;
  float s1, s2, s3, tx, ty;
  int yp, up, vp, tmpr, tmpg, tmpb;

  if (x > 0 && x < iw - 1 && y > 0 && y < ih - 1) {
    s1 = M[0] * x + M[1] * y + M[2];
    s2 = M[3] * x + M[4] * y + M[5];
    s3 = M[6] * x + M[7] * y + M[8];
    tx = std::atan2(-s1, s3) * ((double)panoWidth) / panoFOV + ((double)panoWidth) /
 ↪  2;
    ty = ((double)panoHeight + 500) / 2 - ((double)panoHeight) * s2/std::sqrt(s1 *
 ↪  s1 + s3 * s3);

    //Fetching the texture
    if (tx>=0 && tx<panoWidth && ty>=0 && ty<panoHeight) {
    yp = source[(panoWidth*(int)floor(ty)+(int)floor(tx))];
    up = sourceU[(panoWidth*(int)floor(ty/4)+(int)floor(tx/2))];
    vp = sourceV[(panoWidth*(int)floor(ty/4)+(int)floor(tx/2))];
    tmpb = yp + 1.772*(up-128);
    tmpg =(yp - 0.34414*(up-128) - 0.71414*(vp-128));
    tmpr = yp + 1.402*(vp-128);
    dest[(iw*y+x)*3] = (unsigned char)(tmpb < 0 ? 0 :(tmpb >255 ? 255 : tmpb));
    dest[(iw*y+x)*3+1] =(unsigned char)(tmpg < 0 ? 0 :(tmpg >255 ? 255 : tmpg));
    dest[(iw*y+x)*3+2] =(unsigned char) (tmpr < 0 ? 0 :(tmpr >255 ? 255 : tmpr));

    } else {
      dest[(iw*y+x)*3] = 100;
      dest[(iw*y+x)*3+1] = 0;
      dest[(iw*y+x)*3+2] = 0;
    }
  }
}
```

Listing 3.4: The video frame processing kernel in the first GPU panoramic video viewer

installed in the machine at the time (NVIDIA Geforce 8800 GT) were among the first CUDA enabled graphics cards and only supported CUDA compute capability 1.1. The software uses functions that require a more recent graphics card so we had to equip the machine with a more recent model (NVIDIA Geforce 750 Ti) to solve the issue. For more details on the GPUs used during the writing of this thesis see appendix A.2.

As with the previous prototype this one acts on a locally stored panoramic video with a hard-coded location which requires recompiling should one which to change it. Keyboard controls are used like before, which allows the user to tilt, pan and zoom the camera.

The frame rate is significantly improved with the help of parallel processing and CUDA, running in near real-time at even the highest resolution, however, this is profoundly affected when processing user input, stuttering when a key is pressed and freezing entirely should that key be held. This profoundly effects the user friendliness of the application and makes it difficult to recommend as anything but a proof of concept. The GUI is as spartan as the GUI in the CPU-based prototype. Since we can now use full-size video without scaling the graphical quality is improved, but the implementation suffers from large block-like artifacts when we zoom in as shown in figure 3.7(b). Additionally, while the warping is improved when zooming in on the corners of the field thanks to an automatic tilting of the camera, the lines marking the edges of the midfield are noticeably less straight than they were in the CPU implementation (see figure 3.7(a)).

(a) First GPU-based panoramic video viewer GUI example

(b) GPU-based panoramic video viewer image quality in detail

Figure 3.7: Example images from the first GPU-based panoramic video viewer

3.6 GPU2: Second GPU Prototype

OpenGL textures can be written directly to the screen buffer by the NVIDIA CUDA kernel and a second GPU implementation was created to take advantage of this feature. A texture area is defined in the buffer and once pixel fetching operations complete they are directly written to the texture buffer on the GPU instead of being transferred back to the host. This texture is then displayed on the screen. This process reduces the transfer overhead by bypassing the host. This implementation also uses CUDA texture buffers instead of the GPUs global memory which allows us to use hardware-accelerated interpolation. This speeds up the pixel fetching operation considerably.

3.6.1 Implementation

As with the other prototypes this one has a single entry point in the form of a compiled executable. Arguments may not be passed while executing the program, and are instead hard-coded in the main source file. The arguments needed is the location of the video files to process, as well as the height, width and depth of field of the panoramic video. While the two prior implementations require the path of a local video file this takes either the path of a local or network location, and will process all video files described by a manifest text file at that location in the order specified.

There are multiple modes that the prototype may be run in, and these are changed by altering a conditional directive in the main header file. These modes define whether or not the program is to process videos based on an online and continuously updated manifest file (as in the case of a live soccer game), a finished game stored on a web-server in its entirety, or a set of local files. In the first case the manifest file will be downloaded repeatedly in order to see if a new file has been written yet. If it has, the program will fetch and keep it ready to be processed. In the case of a finished game the program will fetch and process the files described in the manifest file in chronological/lexicographical order. For offline mode we simply read the directory itself and ignore any manifest file.

The videoReader class now explicitly handles both YUV420 and YUV422 color coded video files, although YUV420 is still the preferred and expected format. As for the viewer class one of the bigger changes is that we no longer rely on openCV for our drawing functions. Instead OpenGL functions are used, specifically functions from the **OpenGL Utility Toolkit (GLUT)**. GLUT is a library of utilities for OpenGL programs which is usually used to perform system-level I/O for the host operating system. In this case we create and control the window the program will run, as well as the keyboard functionality.

The viewer class has been expanded significantly to support the new streaming capability, but the technical details of of the viewer implementation is mostly identical to that of the previous GPU implementation. Aside from having to set aside memory to hold incoming streamed video we handle frames in much the same way as before, except this data is now stored on the GPU in a read-only memory space called texture memory. Like constant memory, this texture memory is cached and a texture fetch therefore costs a device memory read only on a cache miss. This cache is optimized for 2D spatial locality, which means that it is particularly suited for operations where a thread is likely to read from an address close to the address read by other threads. Since our kernel has a lot of threads working alongside each other reading a 2D image pixel for pixel this is a great fit.

Another new addition is the possibility of recording a demo video. This functionality is activated by setting a flag in the main header file in the same manner you change between the online/offline modes. If set, the program will use OpenCV data-structures and functions to generate a second virtual view on the host which every finished panoramic frame is copied to and from there on written to disk. This negates most of the performance gain in this prototype, but is a useful function.

3.6.2 Interpolation

Interpolation is a method of creating new data points within a range of discrete data points in a set. We might have a number of data-points and be asked to estimate the value of a new data-point located somewhere between them. In our case we are looking to pin down the value of a new pixel in our virtual view by processing the pixels in the panoramic video, and since the resulting values are usually not perfectly discrete we need to decide on some way to arrive at a compromise. In the two first

implementations of this prototype we have used nearest neighbor interpolation when deciding on the new pixel value, which involves simply making the value the same as that of the closest discrete value (nearest point) without taking the values of neighboring points into account. This is quick and easy to implement, but can result in block-like artifacts and significant loss of information.

The second GPU implementation adds a number of alternative interpolation methods that can be switched between in real time. Nearest neighbor is still an option, but we can also choose between a CUDA-native linear interpolation method and one coded by the author of the prototype. With linear interpolation intermediate points are weighted average of two (or more in some variations) points. The fourth and default option for this implementation is a bicubic interpolation. In image processing, this method is often chosen over bi-linear interpolation or nearest neighbor interpolation in cases where speed is not a big issue. Bicubic interpolation considers a much larger amount of pixels for every one it needs to interpolate (4x4, in contrast to the 2x2 used by bi-linear interpolation), but in return re-sampled images are usually smoother, with fewer interpolation artifacts These alternatives were added to test the visual differences the methods and produced and what the cost would be for each. A comparison performed by Gaddam et al [6] concluded that the bicubic interpolation produces the best results, at a slightly higher performance cost.

3.6.3 User Experience

The prototype follows the same patterns as previous ones and consist of a single executable that needs to be compiled by the user, and which takes no arguments. It runs on UNIX operating systems, and requires all of the libraries the two previous implementations did, but additionally makes use of GLEW and GLUT. Fulfilling the build dependencies and compiling must be done manually. The location of the video to process is still hard-coded, but the program can now act on a general location (local folder or network address) instead of needing the location of a specific video file.

As shown in figure 3.8 the user interface is as spartan as before, but the de-warping now works perfectly across the entire field. Keyboard controls are still the main way of controlling the program, although this prototype demonstrates the possibility of using tracking information gathered by the ZYX subsystem of Bagadus to automatically control the camera. Evaluating this functionality is, however, outside of the scope of this thesis. The frame rate has been slightly improved, but is not noticeably different from that of the previous GPU implementation during normal operation. However, when we use the keyboard controls this causes less stuttering and no longer freezes the video should you hold the zoom button for instance. Other welcome improvements have been made, such as increasing the amount of zoom applied for each press of the corresponding key which makes for more responsive camera controls than in previous incarnations. However, it has become clear how unsuited keyboard controls in general are for controlling the camera in the smooth and responsive fashion you would expect a camera to handle. Analogue controlling methods like a mouse or gamepad would improve the user experience profoundly.

Video quality is improved considerably. Nearest Neighbor interpolation gives us the sharpest images and seems slightly quicker than the others, but produces more artifacts when we zoom in. Figure 3.9 shows a comparison between it and Bicubic interpolation, which is considered the most sophisticated interpolation method of the available ones. Both methods seemed improved over those used in the previous prototype implementations, but we found little discernible difference between the linear and bicubic methods in either image quality or performance.

Figure 3.8: Second GPU-based panoramic video viewer GUI example

(a) Nearest neighbor interpolation

(b) Bicubic interpolation

Figure 3.9: Comparison of interpolation methods used in the second GPU-based panorama viewer

3.7 Summary

In this chapter we have described the first three prototypes that were made of a panoramic video viewer application for use in the Bagadus arena sports analytics system. We began by looking at the design goals and motivation for these prototypes and the panoramic video viewer concept in general, before exploring the evolution of the previous prototypes by examining each in sequence. For each prototype we explained the technologies and techniques used to solve the problem of rendering a virtual camera that allows us to zoom, pan and view the panoramic video without the warping that exist as a result of the panoramic video stitching process. We also briefly described what user experience was like for each prototype.

Clearly a strictly linear implementation making use of the CPU cannot deliver frames quickly enough for real-time operation of the viewer, and moving towards solutions that make use of the GPU make sense. However, our experience also showed that using proprietary technologies such as CUDA for your software limits the user base to those who have a GPU supporting the technology. Additionally, care must be taken when using GPGPU APIs that are written for specific hardware requirements, as those with older cards may also be unable to run the software if they do not support the functionality used in your software.

Compounding the issue of hardware and software requirements issues is the fact that compiling and running these prototypes is too difficult for end users. While work is ongoing to integrate this software into the Bagadus player software we believe there is value in implementing the panoramic video viewer as a web application in order to expose this functionality to end users in a more accessible way. In the next two chapters we present our first attempt at realizing the functionality shown by the prototype using open web standards that could greatly simplify the deployment of this software and potentially reach additional audiences.

Chapter 4

A WebGL-based Panoramic Video Viewer Web Application

4.1 Motivation

The first prototypes described in the previous chapter demonstrated the viability of the virtual viewer idea. Real-time panoramic video with zoom and pan support can be achieved through the use of consumer-grade computer hardware, and the techniques used in the latest implementation are still being used in the latest version of the virtual viewer for Bagadus. This version now supports sophisticated tracking algorithms and can be controlled through an analog game-pad among other extensions to both functionality and user friendliness. It is currently in use at multiple test-sites around the country.

There are however, many areas in which the current implementation have not changed from the early prototypes and which could be improved. The virtual viewer software currently requires a discrete graphics card that supports CUDA. Since this is a NVIDIA proprietary framework, that limits users to those possessing an NVIDIA graphics card, and while NVIDIA is the current market leader with a 76% market share at the end of the fourth quarter of 2014 we are seeing a massive drop in the number of computers who are equipped with these GPUs falling from 63% in the beginning of 2008 to 37% thanks to the weakening of the computer market and ubiquitous **integrated graphics processors (IGPs)** [32]. IGPs are built into 99% of all non-server Intel Processors [33] and are especially common in the budget and mobile markets. Of course, even if you do have a discrete GPU from NVIDIA, not just any card will do, as the viewer is currently written in such a way that only newer cards supporting a later version of the CUDA API will compile properly. Aside from hardware requirements, the software is also dependent on a number of software libraries with their own version requirements. These must usually be acquired and built manually in a way that allows the user to properly compile the virtual viewer. This is a daunting task for anyone but an expert. While this is merely a mild headache during testing and research as developers are usually on hand to install the software and provide support, it is far from ideal when contemplating more widespread deployment. Additionally, while the UNIX family of operating systems are widely available on a number of devices it is far from ubiquitous, and tailoring the software for different devices is a complicated task even without considering alternative operating systems.

Porting the software to a cross-platform computing environment such as Java could go some way to solve the latter problem, but a more flexible solution would be to implement the software as a web application. Web applications have become very popular since web browsers are ubiquitous these days and the use of them as a client make updating and maintaining web applications very convenient. We could solve the both the issue of cross-platform compatibility as well as the issues we currently face regarding the distribution and installing of the virtual viewer on client devices. Additionally, web browsers and the latest web standards now include support for hardware accelerated graphics as a matter of course, and creating an implementation of the virtual viewer that makes use of this would mean that it could be run using a much larger range of graphic processing units.

4.2 Related Work

While there are many **Rich Internet Applications (RIA)** out there these days, true interactive 3D web applications are a relatively new phenomenon. Previous efforts to render 3D in web applications were hampered by a lack of proper GPU integration with the browsers, and developers had to use sophisticated plug-in libraries that enabled 3D rendering using the CPU (such as Papervision3D [34] and Away3D [35]). However, one such project by Dawes et al [36], which set out to combine sports footage with user-interactive 3D graphics, concluded that not only does interactivity engage users to a large degree, but also that this kind of web technology opens up a myriad of possibilities for sports analysis. This is backed up in the book Computer Vision in Sports [37], which concludes that there are still still plenty of problems to solve and new avenues to open in a field with plenty of financial backing.

Foote et al [25] and Pea et al [38] are other examples of panoramic video systems who rely on the same general idea of using a camera array with fixed camera positions, which makes transformations for panorama stitching much easier since you can establish many of the warping parameters offline and then apply the same transformations for every frame of video, much like the Bagadus system did before the implementation of the dynamic image stitcher [39]. Both allow for virtual cameras using similar methods as the ones described in the previous chapter, and both wanted to use the web as a platform. Both solutions were limited by the technology available at the time. The former did create a web page where users could be served a virtual camera that they could interact with, but this did not include actual real-time playback of video, but was instead a webcam set-up where an image would be repeatedly downloaded by the users. The latter project did manage to serve video, but relied on QuickTime, a proprietary viewer plug-in for the browser. They found that writing a custom plug-in that allowed for real-time generation of virtual cameras was too difficult, and instead hosted clips on the web after they had been exported by their software package. Users could thereby view and even comment on the resulting virtual video streams through a web interface, but not create their own without the desktop software.

In 2011, The WebGL 1.0 specification was released and this marked the first integration of a GPU library into the web browser. Browser plug-ins such as Java or Adobe Flash are falling out of favor because of their proprietary nature, restrictive licenses, bad performance and accompanying security vulnerabilities This has resulted in incredible growth and many papers and products using the technology are now available. Google Street View [40] is a widely used panoramic video viewer, and moved away from Adobe Flash to a pure JavaScript implementation, unfortunately it has no video support, and use of the Street View API [41] is limited to non-interactive static images and licenses much be purchased should your solution generate enough traffic. The krpano panoramic video viewer [42] is a commercial product built on Adobe Flash, but added support for HTML5/WebGL panoramic videos with version 1.18, unfortunately the source code is not available. A variety of open source panoramic viewers and tools are indexed and hosted by the PanoTools website [43], but most of these either deal with panoramic images exclusively, are based on outdated browser plug-ins or do not provide the source code. The prototypes we will present differ from those above in that it is built for the specific purpose of using panoramic videos to provide an interactive virtual camera that can cover the entirety of a soccer field and that it will be based on modern and open web standards that take advantage of GPU hardware acceleration

A recent project of particular relevance to us is dos Santos Junior et al [44] which presents an open source WebGL and HTML5-based panoramic video viewer for the web. Their paper additionally compares the performance of their panoramic video viewer with that of the Adobe Flash-based krpano viewer and shows a significant increase in performance at multiple resolutions. Their solution is different from ours in that they use spherical projection and draw their video on the inside of a sphere encompassing the virtual camera, while our WebGL prototype uses a semi-cylindrical canvas facing the camera to reduce the effects of the Homography stitch used by the Bagadus panoramic video pipeline. Their solution, while taking steps to modularize the functionality is geared towards providing immersive tour experiences such as Google Street View using video instead of images, while ours is meant to be used with stored and live streams from a stationary camera array mounted in a sport stadium. Much of the camera controls are similar in these two projects, but we additionally include support for analog gamepads.

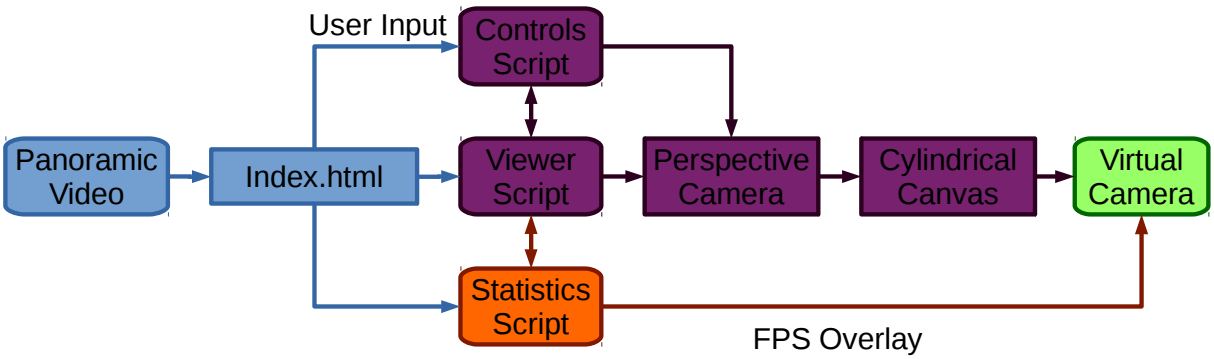


Figure 4.1: WebGL-based panoramic video viewer prototype architecture

4.3 Architecture

We first considered writing a pure port of the GPU implementation for WebGL, but WebGL is not written to be used as a general purpose processing library and we would not easily be able to find replacements for the CUDA functions used in the previous prototypes. We decided we would first prove the general concept of the virtual viewer as a web application and that we would evaluate the performance we could achieve without explicitly programming the parallel nature of the algorithm. Since the general idea behind the panoramic video viewer consists calculating pixel positions of the panoramic video as if they were projected onto a cylinder we came up with an idea for a shortcut that would dispense with the necessity of writing code that did so explicitly. Instead we could create a 3D model for a half-cylinder to serve as a "canvas", and project the panoramic video onto it as a texture. With a virtual camera facing this cylinder at the right angle and with the right field of view, we would achieve the same end-result while relying on standard library functions. Since our idea would make use of very common WebGL functionality we could thus test the viability of our idea as quickly as possible, and our implementation would benefit from any prior and future work in GPU acceleration without requiring any substantial rewrites.

We decided that we would make use the Three.js library as it is one of the most popular helper libraries to offer video-to-texture functionality and perspective cameras. The first prototype would be simple, but should contain all the relevant functionality of the previous ones: It needed to be able to handle both local and remote videos, and should allow the user to control the panning, pitch and zoom of the virtual camera. We also wanted the possibility of monitoring the rendering performance directly while using the prototype, as well as support for a wide variety of control input methods, namely keyboard, mouse and gamepad. In the rest of this section we will detail the underlying technologies used in the WebGL-based panoramic video viewer prototype.

4.3.1 HTML5

The site running the panoramic video viewer script will be written in HTML5 and will take advantage of the video element tag introduced in this version. The HTML5 standard does not specify which video format browsers should support, and unfortunately there is no consensus between the different browser developers on this issue. H.264, which has been used in the Bagadus system thus far, is a proprietary format, and while it remains widely used, and offers good compression, speedy hardware decoding and satisfying video quality there has been controversy surrounding the many patents it relies on. For a time the codec was only supported by the browsers offered by Microsoft and Apple, but it was eventually reluctantly adopted by all major browsers. Threats to pull support has been levied by Google, but ultimately not acted upon. Cisco released an open source implementation of H.264 in 2013 that goes some way to address the problem, but some controversy still surrounds which format to use. We tried using the free Theora codec as used in the .ogg format with limited success, finding the video quality lacking.

A free alternative which has a large audience is the WebM format developed by Google and used in their popular video streaming service YouTube [45]. This format uses the VP8/VP9 codec for video

and provided promising quality and decoding performance in our preliminary testing. The panoramic video viewer itself should remain as format agnostic as possible, and we have using a variety of formats during development to ensure this is the case, but it may be necessary for the Bagadus system, or any other back-end making use of this viewer to serve the panoramic video stream in a variety of formats to maintain universal device coverage.

4.3.2 WebGL

The Web Graphics Library (WebGL) is an API developed and maintained by the Khronos Group for rendering and interacting with 3D and 2D computer graphics from within compatible web browsers. It utilizes the GPU to provide hardware accelerated rendering, image processing, effects and physics, all within the canvas of a web page. It is important to note that WebGL is not a browser plug-in like Adobe Flash that must be installed by the user, but rather an integrated part of the web standard. As such, it is naturally included in most modern browsers. A notable exception is Internet Explorer, which only provides a partial implementation, and is lacking the video texture upload capability that is essential for the implementation described in this chapter.

WebGL is based on OpenGL ES 2.0, and implementation of OpenGL for embedded devices. Code is mostly written in JavaScript, with the exception of shader code which is usually written in a c-like language called the **OpenGL Shading Language (GLSL)**. The API mainly uses an HTML5 canvas element to display graphics in, but WebGL elements can also be combined with other HTML elements, which makes creating user interfaces or compositing interesting scenes easier. WebGL applications can use many convenient parts of the JavaScript infrastructure and the **Document Object Model (DOM)** used in HTML, allowing us to use the browsers image loading functions, event handling and the like. WebGL uses the automatic memory management of JavaScript, which means we no longer have to explicitly allocate and free memory like we have to with OpenGL.

4.3.3 Three.js

Even with the many advantages using WebGL over regular OpenGL provides us, writing sophisticated software with it can still be pretty tedious work. As a result, a number of utility libraries have been developed that can simplify tasks such as setting up view transformation shaders (that define the view frustum), the loading of 3D objects in various formats and the relationships between different objects in our scene. These libraries are usually written in JavaScript

For our WebGL prototype we have used Three.js, a JavaScript library that aims to simplify the development of multimedia web applications. It provides us with many high-level features ranging from scene set-up to cameras, shaders, effects, maths functions, objects, geometry and data loaders. While this would not have been strictly necessary for a simple port of the technique used in the previous prototypes, it comes in extremely handy for the alternative approach as outlined in the next section.

4.3.4 Stats.js

This simple JavaScript class provides us with a info-box overlay that allows us to monitor script performance in real-time It provides FPS and memory use statistics and may be used freely as it is licensed under the open-source MIT license. In section 4.5 we will use it to make an informed decision on whether or not WebGL technology has matured to the point where our prototype is a viable solution for most consumer devices.

4.4 Implementation

The WebGL-based panoramic video viewer web-application prototype consist of a simple HTML5 page and a few JavaScript Currently, we host the website and the example video files on a simple Apache web-server, but the web application could easily be hosted locally, or we could choose to host or stream the video content for a separate location.

4.4.1 HTML Template

We began our implementation by writing the web page our application will run in. For this first prototype a simple HTML5 template is all we need. We remove margins, set the canvas to take up the entire browser window and hide any overflow using in-line **Cascading Style Sheets (CSS)** to maximize the screen area taken up by the panoramic video viewer.

```
<head>
  <title>WebGL Panoramic Video Viewer Prototype 1</title>
    <style>
      body { margin: 0; overflow: hidden; }
      canvas { width: 100%; height: 100%; }
    </style>
</head>
```

Listing 4.1: HTML header with CSS

The example videos are specified in the body of the web page using the HTML5 video element tag. We could just as easily have created the video element and specified the corresponding source file in our JavaScript, but we have opted for doing so in the web page in an effort to make the script as abstract as possible since subsequent projects using this script will likely specify video source files dynamically. The video is set to start playing automatically and loop since we will be using short clips for the testing of this prototype. Since we have specified a video element in the HTML file any HTML5 compatible browser would normally load it is video player, but since our script will be handling the video we have specified that this should be hidden.

```
<body>
  <video id="example_video_1" autoplay loop style="display:none">
    <source src="example.mp4" type='video/mp4' />
    <source src="example.webm" type='video/webm' />
    <source src="example.ogv" type='video/ogg' />
  </video>

  <script type="text/javascript" src="three.min.js"></script>
  <script type="text/javascript" src="viewer.js"></script>
  <script> viewer.init(); </script>
</body>
```

Listing 4.2: HTML body with video element and scripts

A compressed version of the Three.js javascript is also specified in the body, along with the panoramic video viewer script itself. Loading the script does not automatically launch the viewer; it has to be manually started with a call to the init() function. This allows us more fine control over the load and launch order of our scripts, which might come in handy should be wish to extend the panoramic video viewer with additional scripts at a later date.

4.4.2 Initialization

We begun our implementation of the panoramic video viewer script by setting up and initializing the scene, camera and renderer using the Three.js library functions. An element is created to hold the renderer, since it is the only visible element in our page it will take up all of the available space. After initializing the scene we create a camera of the PerspectiveCamera type. Our other alternative here would be an orthogonal camera, using orthographic projection. However, this would not approximate a realistic view of real objects as the relative distance from the camera would have no effect on the size of objects in our scene. The perspective camera takes four argument, the first being vertical field of view for the camera frustum (the pyramid-shaped region of space visible in our scene). For our purposes this will be analogous to camera zoom. We set the aspect ratio of the frustum to be equal to that of our browser window, since we want the panoramic view to fill it. The latter arguments define the clipping planes of

the camera frustum Since we will only be drawing the half-cylinder "canvas" object we really only need to make sure that we define these in such a way as to contain it in it is entirety

The camera is considered an object in it is own right, and must be added to the scene. Once that is done we define the renderer to use, which in this case is the WebGL renderer. Three.js supports a 2D canvas renderer as a fallback for browsers that do not support the WebGL renderer, but this is not recommended for intensive 3D operations. The renderer is given a size equal to that of the browser window and added to our container. This is all that is needed to set up an environment in which to render and view 3D objects.

```
init: function() {
  "use strict";
  var videoMaterial, videoGeometry, videoScreen, radius, height, degreeCount,
  ↪  degrees, faceCount, currentAngle, nextAngle, lastAngle, currentCos, currentSin,
  ↪  nextCos, nextSin, topL, topR, botL, botR;

  container = document.createElement( 'div' );
  document.body.appendChild( container );

  // Initialize scene.
  scene = new THREE.Scene();

  // Initialize camera.
  camera = new THREE.PerspectiveCamera(90, window.innerWidth / window.innerHeight,
  ↪  0.1, 10);
  scene.add(camera);

  // Initialize renderer.
  renderer = new THREE.WebGLRenderer({antialias: false});
  renderer.setSize(window.innerWidth, window.innerHeight);
  container.appendChild(renderer.domElement);
```

Listing 4.3: Initializing the Three.js scene, camera and renderer

4.4.3 Video Texture

The next step was to create a material for the cylindrical canvas we will be creating, using the video data stream as a texture. We fetch the video defined in the HTML file by its element ID and use it as a base for a video textures. In three.js, a video texture is simply shorthand for a texture where the image data (a video in this case) must be continuously monitored for changes, and updated once we have read enough data for a new frame. For video textures we also disable mipmapping, which would normally be used to generate smaller version of the texture for use when seeing the texture from afar. Since we generally want to operate at full resolution, we have decided that storing mipmaps for every frame of video is a waste of resources. The VideoTexture class automatically includes these differences from the regular texture class, but is otherwise identical.

```
// Initialize video.
video = document.getElementById('example_video_1');
videoTexture = new THREE.VideoTexture(video);
videoTexture.magFilter = THREE.LinearFilter;
videoTexture.minFilter = THREE.LinearFilter;
videoMaterial = new THREE.MeshBasicMaterial({map: videoTexture, side:
  ↪  THREE.FrontSide});
```

Listing 4.4: Creating a video texture material

Internet Explorer currently does not support video texture uploads in any form, which is in violation of the standard. It therefore does not support our panoramic video viewer. A bug report has been opened in their bugtracket [46].

4.4.4 Canvas Geometry

In order to display the video texture without the warping inherent to the panoramic video stitching process we need a 3D object with custom geometry. The idea is to create a curved canvas, roughly a half cylinder, and then draw the video texture on the front side of this object. To be more exact, the half-cylinder will consist of 158 degrees of a cylinder, each degree will be represented by two triangular faces (a flat side on the 3D object) who together make up a rectangle. The height of the canvas is set to be 50% more than the radius of the cylinder, but this can easily be changed and the video texture will stretch to accommodate it. We found that 50% results in a better looking panoramic video, even if this does not strictly match the aspect ratio of the video.

We do not use primitives when defining this geometry, instead we manually define the vertices for each corner of a degree of the canvas object, add them to the object in clockwise order, and define two new faces from them which are also added to the object, see code listing 4.5.

```
// Initialize geometry.
videoGeometry = new THREE.Geometry();
radius = 1;
height = 1.5;
degrees = 158;
lastAngle = degrees * 0.01745329252; // 0.01745329252 == 1 degree in radians.
faceCount = 0;

for (degreeCount = 0; degreeCount < degrees; degreeCount++, faceCount += 4) {
  currentAngle = degreeCount * 0.01745329252;
  nextAngle = currentAngle + 0.01745329252;
  currentCos = Math.cos(currentAngle);
  currentSin = Math.sin(currentAngle);
  nextCos = Math.cos(nextAngle);
  nextSin = Math.sin(nextAngle);

  // Create vertices for each corner of the face we are making.
  topL = new THREE.Vector3(radius * currentCos, height * 0.5, radius * currentSin);
  botL = new THREE.Vector3(radius * currentCos, -height * 0.5, radius * currentSin);
  topR = new THREE.Vector3(radius * nextCos, height * 0.5, radius * nextSin);
  botR = new THREE.Vector3(radius * nextCos, -height * 0.5, radius * nextSin);

  // Add vertices in CW order and create two triangle faces from them.
  videoGeometry.vertices.push(topL, topR, botR, botL);
  videoGeometry.faces.push(new THREE.Face3(faceCount + 3, faceCount + 1,
  ↪  faceCount));
  videoGeometry.faces.push(new THREE.Face3(faceCount + 3, faceCount + 2, faceCount +
  ↪  1));
```

Listing 4.5: Creating the canvas geometry

These faces must also be UV mapped, which is the process where we decide how the texture will be projected onto the 3D object. "U" and "V" are the names for the axes of our 2D texture, and are named this way to avoid confusion with the XYZ axes used on the object in the 3D context. If we were working with primitives these texture coordinates would already be a part of the object, but since we are creating a custom geometry they must be added to the object manually as shown in code listing 4.6

These texture coordinates are held in a property called *faceVertexUvs*. We only use the first element *faceVertexUvs[0]* as the others are there mainly for future proofing upcoming versions of Three.js. *faceVertexUvs[0]* contains an array for each face in geometry, and this array is itself an array containing texture coordinates pairs for each of the three vertices in that face. These are given as a 2D vector. These coordinates must be chosen in such a way to correctly map the correct parts of the texture to the geometry. Since we want to treat the panoramic video as if it was already projected onto a cylinder we slice the video texture up into 2*158 slices and give each pair of faces texture coordinates that correspond to a pair of texture slices, as shown in figure 4.2.

Finally, we combine the object geometry and our video texture material into a new mesh object,

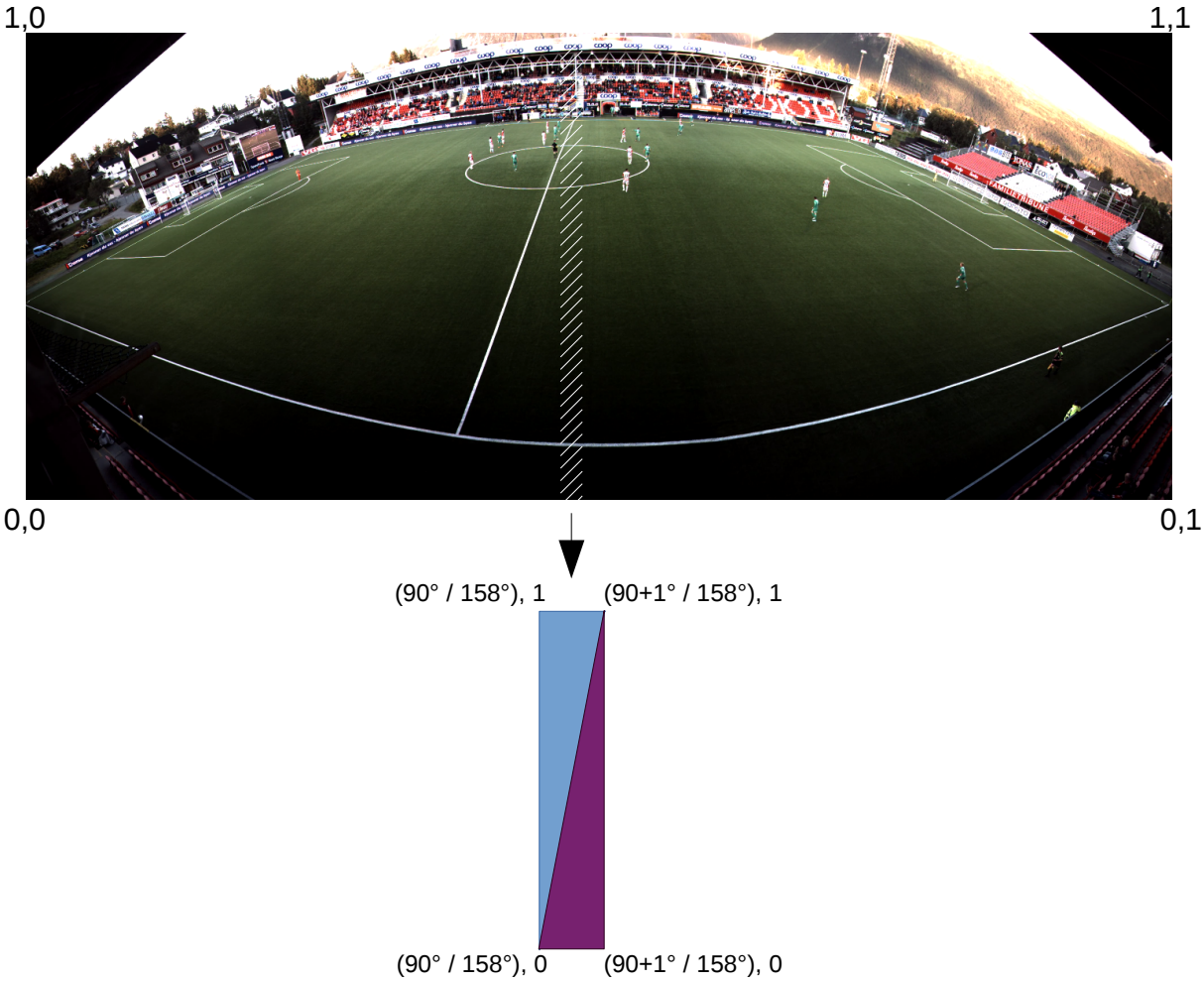


Figure 4.2: UV-mapping of panoramic video texture

```
  // UV map texture across faces.
  videoGeometry.faceVertexUvs[0].push([new THREE.Vector2(currentAngle / lastAngle,
  ↪  0), new THREE.Vector2(nextAngle / lastAngle, 1), new THREE.Vector2(currentAngle
  ↪  / lastAngle, 1)]);
  videoGeometry.faceVertexUvs[0].push([new THREE.Vector2(currentAngle / lastAngle,
  ↪  0), new THREE.Vector2(nextAngle / lastAngle, 0), new THREE.Vector2(nextAngle /
  ↪  lastAngle, 1)]);
}

// Create the mesh, rotate until facing camera and add to scene.
videoScreen = new THREE.Mesh(videoGeometry, videoMaterial);
videoScreen.rotation.set(0, (Math.PI / 2) * (1 + (degrees / 180)), 0);
scene.add(videoScreen);

viewer.update();
```

Listing 4.6: UV-mapping the canvas geometry

rotate the object so the concave side is level and facing our perspective camera, and add it to our scene. This is all that is strictly necessary to display a single frame of unwarped panoramic video, and we invoke the *viewer.update()* function to begin the rendering rendering loop.

4.4.5 Rendering Loop

The rendering loop is implemented by creating a function called *viewer.update()*. We will use this function to ensure that the video texture is updated every time we have read a new frame of video data using the *requestAnimationFrame()* function from the web API. *requestAnimationFrame()* may be implemented slightly differently on various browsers, but the general idea is that the method lets the browser know that you want to animate something and that it should call the function given as an argument before painting the next frame. By having the first line of code in the *update()* function be a call to requestAnimationFrame(viewer.update()) we can ensure that whatever we specify in *update()* should be done before drawing the next frame, and this will continue unless we tell the script to do otherwise. This usually means that we will call the update() function as many times as our refresh rate (generally 60 times per second).

The most important thing we want to ensure happens for every frame is that we render our Three.js scene. Therefore, we make sure to call *renderer.render(scene, camera);* within the *update()* function. Since we are using the VideoTexture class for our texture, we automatically update our texture, but we could just as easily have included a check to see if we have acquired enough data for a new video texture frame and updated it within this function manually. Using requestAnimationFrame() also results in the browser automatically throttling how often it updates should we switch to another tab, which saves resources.

Since the last call in our viewer.init() function is a call to viewer.update() the viewer automatically begins when you open the web page. We have now included enough functionality to draw the panoramic video onto our curved canvas, which when seen through the perspective camera added to our scene results in a wide shot of the field with far less noticeable warping effects as shown in figure 4.3.

4.4.6 User Input

The next step in our implementation is added support for user input. We want our panoramic video viewer to support keyboard controls similar to those of the previous viewers, but we additionally want to provide users with mouse and gamepad support. We believe that these input methods will provide users with more immediate and accurate camera controls, and allow users more closely mimic the smooth and direct movement of an actual camera.

Figure 4.3: Unwarped panoramic video displayed in browser using WebGL-based panoramic video viewer

Keyboard

Keyboard controls are often implemented in one of two ways: event-driven, or state-driven. The former is perhaps the most common to see in web applications and consists of defined event-listeners that fire whenever a button is pressed or released. The latter is a method where the state of the keyboard as a whole is kept updated, so that the developer may write functions that check the current state of any button or combination thereof. For this first WebGL-based prototype we implemented keyboard controls using events listeners as it is the most simple approach, although not the most intuitive, as we would later discover. We proceeded by adding event listeners for the keyDown and keyUp events. What this means is that we asked the browser to look out for whenever a keyboard key was pressed or released, and that it should run a specific function when it detects that events. The adding of event listeners is performed in the *viewer.init()* function as shown in listing 4.7.

```
// Initialize controls and their event handlers
cameraSpeed = 0.01;
deltaZoom = 1;
document.addEventListener('keydown', viewer.keyDown, false);
document.addEventListener('keyup', viewer.keyUp, false);
```

Listing 4.7: Adding event listeners to the panoramic video viewer

We then wrote a function for each event, where a switch statement is used to evaluate which key had been pressed and then performs the corresponding action. We define multiple keys for moving the camera in different directions, and they all work by adding or subtracting a set value called *cameraSpeed* to the *camera.rotation.x, y or z* value. As the renderer is updated we see the resulting rotation. Zooming was initially handled in the same manner. By multiplying the value holding the **field of view (FOV)** of the camera with a value less than 1 we would zoom in, and to zoom out we would multiply with a value larger than 1. Since holding down a key on the keyboard will repeatedly fire keyDown events both of these methods would result in something that approximated smooth camera movement and zoom.

Mouse

Unfortunately, mouse events do not fire repeatedly in the same manner, so holding the mouse buttons would not give us a constant zoom, but would zoom in or out only one step for each click. We were in general not satisfied with the smoothness of the input for the keyboard either, so we looked to augment our solution by storing the respective movement or zoom of the camera in a variable while the button is pressed, and use these to affect the camera at every update of the render. Zooming was handled by a variable named *deltaZoom* that would be set to a value of 1, and which changed to a value greater or lesser than 1 when the keyboard or mouse button responsible for zooming in or out was pressed, and then reset to 1 once the button was released. Keyboard movement was handled in a similar manner by creating a deltaX and deltaY variable that would hold a value that would be 0 or a positive or negative number depending on which direction we wanted to move the camera, by making the value held by this variable changeable by a modifier key we also implemented the functionality of a fine-control option for the keyboard.

Moving the camera with the mouse makes use of the mouseMove event. This event provides read-only properties for the shift in both X and Y coordinates between the current event and the last one, which makes these properties ideal for controlling camera movement as we can simply update the corresponding camera rotation property with the value (after scaling) as shown in listing refmousemove.

```
mouseMove: function(event) {
  "use strict";
  if (moveButton === 1) {                    // If left mouse button is held, update
  ↪  camera rotation based on mouse movement.
    var movementX = event.movementX || event.mozMovementX || 0,
      movementY = event.movementY || event.mozMovementY || 0;
    camera.rotation.y -= movementX * 0.002;
    camera.rotation.x -= movementY * 0.002;
  }
},
```

Listing 4.8: Changing camera rotation based on mouse movement events

Gamepad

One of the more recent additions to the HTML5 web standard is a proper gamepad API. Until recently this functionality was only possible by using elaborate libraries such as gamepad.js [47], but this new specification deprecates most, if not all, of these solutions. The specification [48] is still under development, but most browsers now include most of the functionality described by it, which makes implementing support for a wide variety of gamepads much easier.

Unfortunately, we are still not quite at the point where there is ubiquitous support for this API on all browsers, and some effort must be made to make the script cross-platform compatible. Safari (WebKit) does not support the gamepad API at all, and care must be taken to not rely on gamepad support in the script as a whole. Also, older versions of Chrome do not support gamepad connection events, and gamepads must be polled manually using the *navigator.webkitGetGamepads()* function. We decided to implement the gamepad support in a way which should support most browsers, and which is easy to read and understand. We will simply poll all gamepads connected to the computer in every *viewer.update()* call. This approach is not the most efficient, nor is it the most sophisticated way of using the gamepad API, but for the purposes of our demonstration it should suffice.

As shown in listing 4.9 we start by requesting a list of all connected gamepads from the browser using either the standard function call, or the legacy fallback for older versions of chrome. We then loops through this list, checking whether the gamepads within are defined or not. This is not strictly necessary for Firefox as their handling of connection events ensure that this list is properly populated, but the Chrome browser will return a list with undefined objects so we are doing it for compatibility reasons. For each gamepad we update a separate set of delta variables, as we want the gamepad to coexist with the rest of our implementation. Since we are updating the gamepad all the time the values of the other

input methods would simply get rewritten.

Each gamepad object from the gamepads list gives us the status of that gamepad, which includes all of it is buttons and axes. On Firefox these values are usually kept updated at all times, but for Chrome these will be the status of the gamepad at the last *getGamepads()* call. Since we poll the gamepads every update this presents us with no practical difference, so we check the status of the buttons we want to use and update the *padDeltaZoom* variable accordingly. The gamepad axes are very sensitive so we divide their value to make the camera handling easier for the user. At this point we also invert the axes as this makes for more intuitive controls. A problem we ran into is that there does not seem to be any dead zone configured for the browser or the gamepad controller, and the camera would drift resulting from minute offsets on the axis. We solved this problem by clamping any minor input to zero.

```
// Find connected gamepads, if supported by browser.
gamepads = navigator.getGamepads ? navigator.getGamepads() :
  ↪  (navigator.webkitGetGamepads ? navigator.webkitGetGamepads() : null);
if (gamepads) {

for (var i = 0; i < gamepads.length && gamepads[i]; ++i) {
  var pad = gamepads[i];

  // Update Zoom
  if (pad.buttons[0].pressed) {
    padDeltaZoom = 0.95;
  } else if (pad.buttons[1].pressed) {
    padDeltaZoom = 1.05;
  } else {
    padDeltaZoom = 1;
  }

  // Adjust gamepad sensitivity
  padDeltaX = pad.axes[1] * -0.02;
  padDeltaY = pad.axes[0] * -0.02;

  // Adjust dead zone
  if (padDeltaX < 0.005 && padDeltaX > -0.005) padDeltaX = 0;
  if (padDeltaY < 0.005 && padDeltaY > -0.005) padDeltaY = 0;
}
```

Listing 4.9: Fetching gamepad information in viewer.update() function

As shown in listing 4.10 we had to rewrite the way we update the camera rotation and zoom so that it makes use of any combination of the input methods described in this section. This preserves the smooth movement we expect from all of them while allowing neither to completely interrupt input from the others. Using multiple input methods at once is also possible.

A feature that was added in the latest CUDA enabled panoramic video viewer prototype was a slight tilt as the camera moved towards the left and right edges of the stadium. This compensates for the elevation differences in our camera array, and we wanted to add it to the browser as well. Unfortunately, the formula for how this is to be done is very much dependent on the set-up for each stadium and we must provide this with the video file if the feature is to be used. For this prototype we included a tilt that suits videos from Alfheim Stadium, as this is where our demonstration videos were recorded. We also added a zoom limit which ensures that users will not set the FOV to values that exceed what is necessary for normal operation.

4.4.7 Resizing

The user might want to resize the browser window, and we need to handle this event or the user will end up with a distorted render. To do this we add an event handler to the script during the *viewer.init()* function, which checks for this occurrence and calls a function we have written called *windowResize()*; This function adjust the size of the renderer to the new browser window parameters and then does

```
// Adjust camera movement.
camera.rotation.x = camera.rotation.x + padDeltaX + deltaX;
camera.rotation.y = camera.rotation.y + padDeltaY + deltaY;

// Stadium edge tilting of camera.
camera.rotation.z = 0.32 * (camera.rotation.y - 0.027);

//Adjust camera zoom
var newZoom = (deltaZoom * padDeltaZoom);

//Limit zoom depth.
if ((camera.fov > 10 && newZoom === 0.95)||(camera.fov < 90 && newZoom === 1.05)) {
  camera.fov *= newZoom;
  camera.updateProjectionMatrix();
}
```

Listing 4.10: Updating the camera rotation and zoom using the gamepad and other input methods

the same to the aspect ratio of the perspective camera in order to compensate. This also requires and additional call to the cameras updateProjectionMatrix function.

4.4.8 Statistics

In order to be able to gauge the performance of our WebGL enabled prototype we wanted a visible indicator of FPS. Chrome includes a screen overlay which is capable of this in its browser developer tools, but we wanted something that would be identical on every Browser and which we could easily customize Our main concern was that we wanted to be able to isolate the performance information gathering to just the rendering of our virtual camera. By using the open source javascript called stats.js we could specify a start and stop point for the performance timing. To make use of this functionality we added the script to our main HTML file (index.html) and initialized the statistics widget in our *viewer.init()* function as shown in listing 4.11. This involves creating a new *Stats()* object with a global variable pointing to it. We also define some style elements for it, and append it to the main document body so that the widget will appear in the top left corner of the browser.

```
// Initialize statistics widget.
stats = new Stats();
stats.domElement.style.position = 'absolute';
stats.domElement.style.top = '0px';
document.body.appendChild(stats.domElement);
```

Listing 4.11: Adding the statistics widget to the viewer

We can then call the *stats.begin()* function when we want to start timing performance, and then call *stats.end()* at the point where we want it to end. By placing these function calls before and after the *renderer.render()* call in the *viewer.update()* function we hope to ensure the most accurate timing of the actual panoramic video drawing.

4.5 Evaluation

In this section we will evaluate the WebGL-enabled panoramic video viewer prototype, and directly compare it to the prototypes that came before it. We will then evaluate how the WebGL-enabled prototype performs on a variety of different platforms, equipped with a variety of browsers and GPUs.

4.5.1 Video Quality

In order to test the video quality of our new prototype we concatenated and re-encoded files from the data-set described in section 3.2, using a build of FFMPEG compiled with support for the VP8 and H.264

(a) Second CUDA-enabled panoramic video viewer example

(b) WebGL-based panoramic video viewer example

Figure 4.4: Comparison of linear filter and graphic detail in CUDA and WebGL based panoramic video viewers

codec.

Encoding WebM files with the LibVPX library turned out to be very slow, which could very well disqualify it as an alternative for live video streaming. The reason we are re-encoding is so we may have a long seamless video file to work with, as the WebGL-based panoramic video viewer does not itself have the capability of reading the manifest file at this time. This may be implemented later, or it might be handled by a separate part of the web application backbone.

The video quality of the panoramic video appears to be of equivalent quality to that of the latest CUDA-enabled prototype. As shown in figure 4.5 and figure 4.6 the night-time panorama displayed by our WebGL-based prototype in figure 4.6(b) is slightly darker than that of the latest CUDA-based prototype which is shown in figure 4.6(a).

Both of these implementations use a linear filter to interpolate pixel values and the results appear to be comparable as shown in figure 4.4(a) (CUDA) and 4.4(b) (WebGL). In this figure we can also see the slight degradation in quality that occurs from re-encoding the H.264 encoded source files into the VP8 encoded WebM format. It is our belief that we should stick with the H.264 format as default, as long as that is the format which the pipeline produces, while possibly providing WebM files as an alternative format for stored panoramic videos to serve those who do not support the H.264 format.

The de-warping process does appear to give results that are comparable to the latest CUDA prototype, although further tweaking of the canvas geometry could produce even better results. We can also clearly see that the minute differences in the way these panoramic video viewers are configured greatly affect the de-warping process. While the WebGL-based prototype gives better results than the first two prototypes the second CUDA-enabled prototype has a slight edge in the de-warping of this particular panoramic video. This is a flaw all of the prototypes share; any change in the set-up of the recording array or panoramic video generation pipeline may impact their performance. The aspect ratio of the incoming video and the position of the cameras must remain constant factors and any deviation results in non-

(a) Panorama video from CPU-based prototype

(b) Panorama video from first CUDA-based prototype

Figure 4.5: Night-time panoramic video comparison (part I)

(a) Panorama video from second CUDA-based prototype

(b) Panorama video from WebGL-based prototype.

Figure 4.6: Night-time panoramic video comparison (part II)

Figure 4.7: Day-time panoramic video example using WebGL-based panoramic video viewer

perfect de-warping For example, compare the results in figure 4.6(b) (resolution: 4450x2000, aspect ratio: 2.23) with the results we got in figure 4.7 (resolution: 4096x1680, aspect ratio 2.44). The WebGL prototype was developed using mainly this daytime video and the de-warping is more complete in this case.

4.5.2 User Experience

Comparing this new WebGL implementation to the earlier prototypes one thing that is immediately clear is that the solution is more accessible. A user only has to point their browser to a particular URL and the browser automatically downloads all necessary libraries either from the site itself or through **Content Delivery Network (CDN)** servers. There is no need for users to concern themselves with installing or configuration any software, nor do they require more security permissions than those necessary to use a javascript-enabled browser in order to make use of the software. Similarly, when the developers or maintainers of the software wishes to update the software they only have to make changes to the script at the source and the changes will go into effect for all users the next time they run the software. Costs related to deployment and maintenance are potentially drastically lowered.

The addition of gamepad and mouse controls greatly increase the responsiveness of the panoramic video viewer user interface, and allows for smooth pans with the gamepad and near instant shifts in focus with the mouse. The keyboard controls are also improved, with a more intuitive set of default key-bindings, faster and smoother camera movement and a modifier key for fine control of movement for use when the camera is zoomed very far in. The immediate impression we got on our development machine was that the panoramic video viewer prototype is responsive and runs in at least near real-time. We will explore the actual performance in the next section.

4.6 Performance

We ran the WebGL-based prototype on a number of different machines (described in detail in chapter A) in order to first of all see if the panoramic video viewer would run as intended on a wide variety of platforms, and then get a rough estimate of the performance we might expect using non-scaled high resolution video. Using the included statistics widget we wanted to estimate the frame rate our panoramic video viewer would run at, and what limitations that might set for the video frame rate we could support. Ideally we would like the panoramic video viewer to support the newer 50 FPS video used in the latest panoramic video pipeline, with 30 FPS video being an acceptable alternative.

4.6.1 Video Texture Uploads

When development of the WebGL prototype began the panoramic video viewer prototype was surprisingly slow, and not anywhere near 30 FPS on any of our test machines. In order to figure out what the problem was we used the profiling tools included in the Chrome web-browser, and as shown in figure 4.8 we can see that by far the most intensive part of the program is the function call *texImage2d()*, which takes up ∼90% of all running time. This function call is a part of the browsers WebGL implementation so we researched the problem and it turns out we were far from the only people experiencing it. Multiple bug reports [49–53] have been filed because of the issue across all the browsers, with color space conversion being done on the CPU rather than the GPU being identified as the main culprit, but little work was done to fix it until mid-2014 when progress was made by the Firefox developers. We started noticing speed improvements on both the Chrome and Firefox browsers after that as some of the work was ported over.

While work on improving the WebGL implementation on all browsers is still ongoing, we wanted to ascertain whether or not each browser is at a point where we can feasibly roll out a system based on our WebGL panoramic video viewer prototype. We ran two online WebGL video texture upload tests [54,55] designed to gauge progress on just this issue on all of our test machines.

The first test plays an mp4 file encoded with the H.264 codec with a resolution of 1920x960, uploads the frames to a WebGL texture and performs a simple hue-rotation effect to ensure us that the video is running in a WebGL context. This test uses the texImage2d function call exclusively to upload textures. The second test plays a video in resolutions ranging from 480p to 1080p using both the texImage2d function call and the related texSubImage2d call (a function which allows you to upload parts of a texture, something which we do not currently make us of). It does this for the H.264 codec, the VP8 codec and the Theora codec.

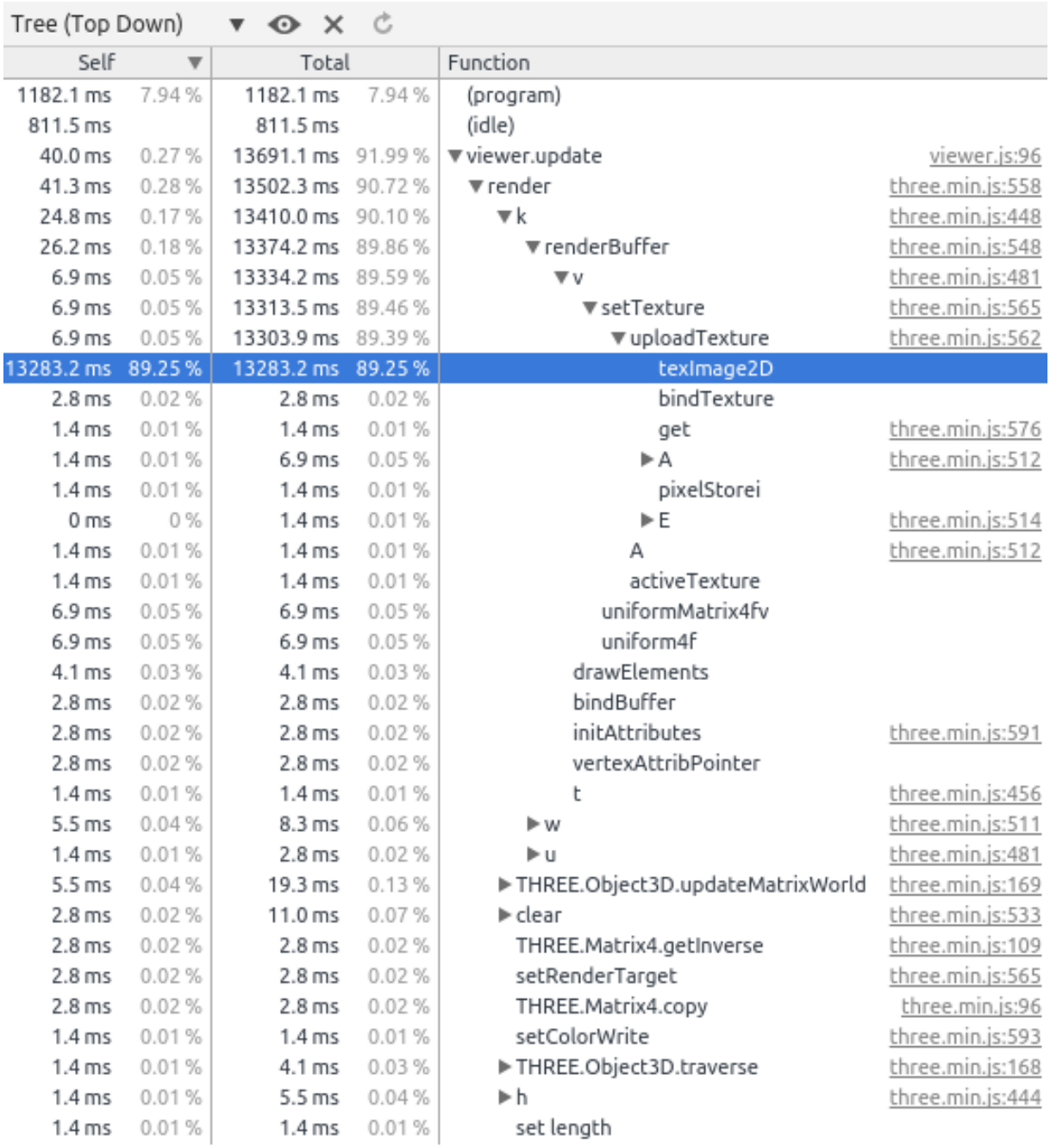
The results of these tests are summarized in table 4.1, and the full tables of performance data for the second test may be found in appendix B. Unsurprisingly we see that the larger the video is, the longer it takes to upload one of the video frames as a texture. Performance appears to be greatly dependent on the browser it is being run on, with Safari clearly struggling the most with video texture uploads. Safari would also only support mp4 files encoded with the H.264 codec.

While Firefox is leading the charge in addressing these problems we found that performance was still greatly dependent on the underlying OS and hardware, and while it appeared to have a slight edge for all the machines equipped with discrete GPUs it was slower than Chrome on the laptop test machine (Mi-go) equipped with an IGP. Surprisingly, using the Chrome browser we achieved performance that beat out the other computer running Linux (Hastur), even though it is equipped with a modern GPU. From all of this we can conclude that the WebGL implementations that currently reside in browsers are very inconsistent in the performance they can provide as far as video textures are concerned, and that (perhaps because of poor optimization) the GPU is not wholly the determining factor as far as performance goes. We should be able to achieve workable performance on most browsers at this time if we keep video resolutions low, but further improvements must be made to handle the 4K+ resolutions which we would like to use.

4.6.2 WebGL Panoramic Video Viewer

Our performance testing of the WebGL-based panoramic video viewer prototype yielded mixed results as shown in table 4.2. There was little difference between simply playing the video files and using the WebGL-based panoramic video viewer, so at the very least we know that the WebGL panoramic video viewer implementation is sound, but working with large video files is an area where browsers must improve their code. WebM performance was particularly bad in Firefox across all of our test computers, resulting in intermittent freezing with the new media reader code, and constant stuttering with the old (switchable in the advanced configuration file of the nightly Firefox build).

For now, the WebGL-based panoramic video viewer appears to works best using the H.264 format on the Chrome browser, using a powerful current generation computer. A discrete GPU appears to be a good thing, but having a powerful CPU appears to be at least as important judging from the results we achieved on our development machine (An old computer with a modern GPU installed), as the results we achieved using the Mi-go laptop test machine were surprisingly good in the Chrome browser.



Tree (Top Down)

Self		Total		Function	
1182.1 ms	7.94 %	1182.1 ms	7.94 %	(program)	
811.5 ms		811.5 ms		(idle)	
40.0 ms	0.27 %	13691.1 ms	91.99 %	▼viewer.update	viewer.js:96
41.3 ms	0.28 %	13502.3 ms	90.72 %	▼render	three.min.js:558
24.8 ms	0.17 %	13410.0 ms	90.10 %	▼k	three.min.js:448
26.2 ms	0.18 %	13374.2 ms	89.86 %	▼renderBuffer	three.min.js:548
6.9 ms	0.05 %	13334.2 ms	89.59 %	▼v	three.min.js:481
6.9 ms	0.05 %	13313.5 ms	89.46 %	▼setTexture	three.min.js:565
6.9 ms	0.05 %	13303.9 ms	89.39 %	▼uploadTexture	three.min.js:562
13283.2 ms	89.25 %	13283.2 ms	89.25 %	texImage2D	
2.8 ms	0.02 %	2.8 ms	0.02 %	bindTexture	
1.4 ms	0.01 %	1.4 ms	0.01 %	get	three.min.js:576
1.4 ms	0.01 %	6.9 ms	0.05 %	▶A	three.min.js:512
1.4 ms	0.01 %	1.4 ms	0.01 %	pixelStorei	
0 ms	0 %	1.4 ms	0.01 %	▶E	three.min.js:514
1.4 ms	0.01 %	1.4 ms	0.01 %	A	three.min.js:512
1.4 ms	0.01 %	1.4 ms	0.01 %	activeTexture	
6.9 ms	0.05 %	6.9 ms	0.05 %	uniformMatrix4fv	
6.9 ms	0.05 %	6.9 ms	0.05 %	uniform4f	
4.1 ms	0.03 %	4.1 ms	0.03 %	drawElements	
2.8 ms	0.02 %	2.8 ms	0.02 %	bindBuffer	
2.8 ms	0.02 %	2.8 ms	0.02 %	initAttributes	three.min.js:591
2.8 ms	0.02 %	2.8 ms	0.02 %	vertexAttribPointer	
1.4 ms	0.01 %	1.4 ms	0.01 %	t	three.min.js:456
5.5 ms	0.04 %	8.3 ms	0.06 %	▶w	three.min.js:511
1.4 ms	0.01 %	2.8 ms	0.02 %	▶u	three.min.js:481
5.5 ms	0.04 %	19.3 ms	0.13 %	▶THREE.Object3D.updateMatrixWorld	three.min.js:169
2.8 ms	0.02 %	11.0 ms	0.07 %	▶clear	three.min.js:533
2.8 ms	0.02 %	2.8 ms	0.02 %	THREE.Matrix4.getInverse	three.min.js:109
2.8 ms	0.02 %	2.8 ms	0.02 %	setRenderTarget	three.min.js:565
2.8 ms	0.02 %	2.8 ms	0.02 %	THREE.Matrix4.copy	three.min.js:96
1.4 ms	0.01 %	1.4 ms	0.01 %	setColorWrite	three.min.js:593
1.4 ms	0.01 %	4.1 ms	0.03 %	▶THREE.Object3D.traverse	three.min.js:168
1.4 ms	0.01 %	5.5 ms	0.04 %	▶h	three.min.js:444
1.4 ms	0.01 %	1.4 ms	0.01 %	set length	

Figure 4.8: Performance profiling of WebGL-based panoramic video viewer

Machine	Browser	Test 1 (1920x960 H.264 Video)	Test 2 (1920x1080 H.264 Video)
Hastur(A.1)	Firefox	∼60 FPS	∼42 FPS
	Chrome	∼60 FPS	∼28 FPS
Azatoth(A.2)	Firefox	∼59 FPS	∼59 FPS
	Chrome	∼59 FPS	∼59 FPS
Nodens(A.4)	Safari	∼30 FPS	∼36 FPS
	Chrome	∼59 FPS	∼48 FPS
Mi-go(A.3)	Firefox	∼60 FPS	∼31 FPS
	Chrome	∼60 FPS	∼59 FPS

Table 4.1: Performance testing HTML5 video playback using WebGL textures

Machine	Browser	4096x1680 VP8 @ 50 FPS	4450x2000 H.264 @ 25 FPS
Hastur(A.1)	Firefox	∼22 FPS	∼12 FPS
	Chrome	∼30 FPS	∼22 FPS
Azatoth(A.2)	Firefox	∼24 FPS	∼55 FPS
	Chrome	∼59 FPS	∼59 FPS
Nodens(A.4)	Safari	∼N/A	∼21 FPS
	Chrome	∼23 FPS	∼51 FPS
Mi-go(A.3)	Firefox	∼16 FPS	∼22 FPS
	Chrome	∼32 FPS	∼59 FPS

Table 4.2: Performance testing the WebGL-based panoramic video viewer prototype

4.7 Future Work

In this section we will present a few issues we have identified with this new WebGL-based panoramic video viewer prototype, and we will propose a few potential improvements that can be made. One major issue is the use of configuration settings that must be manually discovered for each site where we want to deploy the Bagadus camera array, and then hard-coded into the script. Making these changes are relatively easy since we do not have to compile and redeploy the system whenever we make a change, but a more generalized system which was agnostic to this information and simply received it with the video stream, or discovered the necessary parameters from the video information would be ideal. This would require a much more sophisticated and malleable canvas object, as well as a working back-end for delivering the video streams and meta data to users, neither of which time allowed for us to develop for this thesis.

An experimental feature in the CUDA-based panoramic video viewer prototype is ball and player tracking, and we would like to carry this functionality over to the WebGL-based prototype as well. This would require parsing and processing the meta data provided by the ZYX sensors the players wear, but would greatly increase the easy of use of the viewer. Users could simply click a player and the camera would automatically focus and follow them.

Another idea is to port the panoramic video viewer to the new WebCL W3C specification [8]. WebCL is a JavaScript binding for the OpenCL library for parallel computing. This new standard aims to allow any browser to make use of multi core CPUs and GPUs without having to rely on plug-ins. However, no browser has yet even expressed much interest in implementing this standard, and since the performance issues we do experience with our WebGL prototype appears to be mostly related to browser-specific implementation of WebGL it is difficult to rationalize the need unless additional features that require a lot of computation is added to the panoramic video viewer.

Finally, while integrating this WebGL-based panoramic video viewer into the existing Bagadus web-tools would be one way of using it, we think that it might be worthwhile to extend the viewer and create a more fully featured video player application around it. Sport fans would serve as a possible target audience for this application, and we envision them both watching games live and sharing video clips of their favorite games using a video player with the panoramic video functionality built in. The experience of watching games live could be enriched by the panoramic video viewer feature, allowing fans to directly engage with the game they are watching by pressing a button which switches them from the one-camera video stream to a panoramic video stream provided by the Bagadus panoramic video pipeline, effectively taking control of the camera to focus on what they are most interested in.

4.8 Summary

In this chapter we have presented our WebGL-based panoramic video viewer prototype. We first looked at the motivation behind this idea and especially how recent advances in web and graphics technology

make GPU-accelerated multimedia web application a worthwhile topic for study. We then looked at some earlier attempts at similar solutions which solidified the decision to use WebGL for our prototype. This was followed by a presentation of the general design of our panoramic video viewer and the underlying technologies. The key idea here was the use of Three.js javascript library functions to create and interact with a perspective camera observing a curved half-cylinder canvas onto which we project a video texture. This allowed us take advantage of GPU accelerated browser functions to perform complicated 3D transformations resulting in a working panoramic video viewer prototype without relying on proprietary GPGPU technologies or bulky browser plug-ins.

The implementation of the prototype for this WebGL-based panoramic video viewer was detailed next, and we here gave a sense of how the prototype took shape and what influenced the decisions we made during the development process. In addition to replicating the functionality of previous panoramic video viewer prototypes we expanded upon them by implementing new and improved input methods such as mouse and gamepad support. This resulted in smoother and more responsive controls across the board, with keyboard controls also being more flexible and intuitive.

This was followed by a presentation of experiments we carried out to ascertain the performance of in-browser video playback and video texture uploads, which we found to be inconsistent and highly dependent on the browser, hardware and operating system in question as shown in table 4.1, but workable and currently being improved. We also presented the experiments we carried out on the WebGL-based panoramic video viewer prototype itself, and while issues inherent to the WebGL video texture upload function and outside our control kept us from achieving perfect results on all test machines the results were nevertheless very promising, with good frame rates for very high resolution video streams even on mobile computers as shown in table 4.2.

Finally, we presented some future work that could be done based on this prototype in order to provide better de-warping quality, add new features, potentially improve performance, and reach new target audiences. This leads into the next chapter of this thesis where we will seek to improve upon our WebGL-based panoramic video viewer prototype by including functionality that has thus far been missing from any prior panoramic video viewer implementation.

Chapter 5

Improving the WebGL-based Video Panoramic Video Viewer Prototype

5.1 Motivation

A number of improvements came to mind while implementing the WebGL-based panoramic video viewer, and we will present the work that was done to implement them in this chapter. The lack of video controls has been a weakness in all of the previous prototypes and implementing it would allow users to pause the action and skip forwards and backwards in time when watching stored videos. These are common features that are important when reviewing video footage. By laying the groundwork for a more fully featured GUI we can increase the user-friendliness of the web application considerably.

While we are hopeful that support for high-resolution video files in browsers will continue to improve, it might be worth exploring a variety of scaling techniques for the panoramic video viewer, both to increase performance and reduce bandwidth consumption. For example, even when zoomed out into an overview of the entire field many computer monitors do not have the pixel real estate to display the video unscaled and we would lose little graphic detail by scaling the video before we even transmit it to the client. Zooming in would necessitate the full resolution if we want to take advantage of the full image quality the camera array can provide, but it might be possible to segment the video stream itself and have the server stream only the region of interest as was described by Mavlankar et al [56].

Likewise, switching between video sources of varying quality could improve performance and reduce bandwidth usage. The former is still a concern for slower computers with the high resolution videos that the Bagadus panoramic video pipeline produces, and the latter would be especially interesting should we wish to use this technology to reach a large number of users, such as soccer fans streaming from home. By streaming a lower quality stream when zoomed out, and switching to a higher quality stream when we zoom in and need more graphical fidelity we can achieve both.

Another somewhat more complicated solution to the same problem would be to segment the video into multiple regions that correspond to areas off our canvas, and then have the client decide which streams it will pull data from based on which segments are currently within the view of the camera.

5.2 Video Controls

The lack of video controls has been a weakness in all of the previous prototypes and implementing it would allow users to pause the action and skip forwards and backwards in time when watching stored videos. Laying the groundwork for a more fully featured GUI would increase use user friendliness of the web application considerably.

We tried a few different approaches for extending the panoramic video viewer with video controls and we will detail the ones with merits in this section. For demonstration purposes we will focus on the implementation of those controls we believe would be most useful within the Bagadus context, namely time skipping (seeking) and pause functionality.

5.2.1 Using the Default HTML5 Video Player

Our first solution was to attempt to integrate the WebGL-based panoramic video viewer into the default HTML5 video player that the browser uses. We created a new container for the video player in which we include the <*video*> tag with a *controls* option tag for in order to display the video player itself with the default controls. We then added a new button object as a child to the container and positioned it with CSS so it seemed to integrate with the default controls.

We then wrote a javascript with an event listener for this button, which when pressed would hide the video player, initialize the panoramic video viewer, store and continue playing the video in its place. This worked well, however, once we were in the panoramic video viewer context the video controls native to the video tag would no longer display since they were bound to the element we just hid. In order to fix this problem, we decided to create custom controls specific for our panoramic video viewer instead. This will ensure that we can fully control the behavior and look of these controls so that they are consistent across all browsers. Extracting how far into the video a user has gotten (time code) is easily done as this information is one of the attributes of the video element. This makes switching from one video source to another and continuing from the same point easy, so switching from a one-camera stream to a panoramic video stream should be no issue either, as long as the videos are synced, or come with the necessary meta-data.

5.2.2 Adding Custom Video Controls to the Panoramic Video Viewer

We started by adding a play/pause button and a seek bar to the panoramic video viewer as shown in listing 5.1, and positioned the *#video-controls* element in the top-right corner through CSS. These controls can also be used on the default video player should one wish to use it, but significant CSS styling is necessary to achieve pretty results across the different browsers. We decided to focus mainly on the WebGL-based prototype itself.

```
<div id="video-controls">
  <button type="button" id="play-pause">Pause</button>
  <input type="range" id="seek-bar" value="0">
</div>
```

Listing 5.1: Adding video controls to WebGL-based panoramic video viewer

In order for the new control elements to be functional we had to add event listeners to our javascript for each element, and then write functions that implement the functionality that we expect. The event listeners are added to panoramic video viewer in the *viewer.init()* function in the same manner as was previously shown in listing 4.7. The pause button uses a event listener that fires when the button is clicked and the corresponding *viewer.clickPlay()* function alternatively pauses and starts the playback of the video. The seek bar also calls this function for both *mouseDown* and *mouseUp* events, which keeps the video from continuing to play as we seek. The seek bar also has event listeners for *change* and *timeupdate* events. The former fires whenever we seek to a new position and is handled by the function *viewer.changeSeek()*, which calculates the new time code for the video based on the new seek bar value. The latter is periodically fired by the video element itself as the video progresses, and the corresponding *viewer.timeUpdate()* function updates the seek bar value to illustrate the progress of the video.

We also styled the controls using CSS so they fade in and out when the mouse pointer hovers over the body of the WebGL-based panoramic video viewer. This allows users to remove the controls by placing the mouse pointer elsewhere to remove them from view during video playback.

5.3 Zoom-based Adaptive Quality

Since the performance of the WebGL-based panoramic video viewer is greatly reduced when using large textures we experimented with the idea of scaling the resolution of the input video. Another consideration is bandwidth use. While streaming large files are becoming less of a problem for client One popular approach is adaptive bit-rate streaming, which is a technique were the bandwidth and processing

capabilities of the user is detected and adjusted for in real time by reducing or increased the quality of the video stream. This requires an encoder which is capable of encoding a single video source into multiple bit rates, and clients simply switch between the various encodings based on the resources that are available. Each bit rate stream is usually segmented into small multi-second parts and clients read an available manifest file that carries information about which streams are available for which segments. This is similar to the way Bagadus currently streams content.

We wanted to observe whether changes in video source could be done based on zoom level in order to make use of lower quality video when zoomed so far out that the video information is likely have been lost anyway. In order to do this we encoded an example video file in the WebM format with the first being encoded in 4096x2000 resolution and a bit-rate of approximately 10000kb/s, and the second being a version scaled down to 2048x840 and a bit-rate of approximately 1000kb/s. We added both as sources in our <*video*> tag, starting with the low quality encode as our viewer begins in a zoomed out view.

Adding support for switching between two video sources in the script was done by checking the current FOV of the camera during the viewer.update() call which is run for every frame of animation in the panoramic video viewer web application. Whenever the FOV crosses a certain threshold the script checks to see if it is currently displaying the low or high quality stream and if the stream it is displaying is unsuited for the current zoom level it switches the video source as shown in listing 5.2. We save the time code of the first video source in a variable before the switch is performed so we may resume the video where we left of after switching the video source.

```
if (camera.fov < 50 && lowRes) {
  video.pause();
  oldTime = video.currentTime;
  video.setAttribute('src', 'example30hi.webm');
  video.load();
  video.currentTime = oldTime;
  video.play();
  lowRes = false;
} else if (camera.fov >= 50 && !lowRes) {
  video.pause();
  oldTime = video.currentTime;
  video.setAttribute('src', 'example30lo.webm');
  video.load();
  video.currentTime = oldTime;
  video.play();
  lowRes = true;
}
```

Listing 5.2: Switching between high and low quality video sources based on zoom level/camera FOV

Realistically, we have to either preload multiple streams (which would be counter-intuitive as far as bandwidth is concerned) or use a more sophisticated implementation of adaptive bit-rate streaming such as Apple HTTP Live Streaming [57], MPEG-DAS [58], or Microsoft Smooth Streaming [59] protocols. Until recently this would have required bulky browser plug-ins such as Adobe Flash or Microsoft Silverlight, but the new Media Source Extensions W3C specification [60] allows JavaScript to generate media streams for browsers that support HTML5 video. This opens up for implementations using adaptive streaming, time shifting, client-side prefetching and advanced buffering techniques entirely in JavaScript.

5.4 Canvas Segmentation

We wanted to explore whether segmenting the canvas and streaming to only those segments that are currently being looked at by the user is possible, and also whether or not this would increase or decrease the performance of our virtual viewer. We therefore sat out to implement a proof of concept where we split the canvas geometry in half vertically and draw one video texture for each half.

We first split an example video in the WebM format by using the video filter functionality in the

FFMPEG software package. The **crop** keyword allowed us to specify the range of pixels we wanted to re-encode, and by specifying the width of the video file divided by two we produced one video for the left side of the field and one for the right. These were added to the main HTML document as separate HTML5 video elements and autoplay was disabled so we could ensure the best possible synchronization by starting both videos during the script.

Next we modified the generation of our half-cylinder canvas geometry so that we had two quarter-cylinder canvas geometries. This involved halting the creation of vertices at the halfway point for the first object and then continuing from that point for the geometry of the second canvas object. As we had effectively halved the span that our textures would have to be mapped to, these mappings had to be scaled so that they would stretch based on their new geometries. Otherwise the approach is identical to the one described in listing 4.5 (access the full source code as described in appendix C to see changes in detail).

The *viewer.update()* function also had to be extended, with a second check for the extra video source that informs the renderer when it needs to update the video texture for the second part of the canvas. In order to demonstrate the idea of this approach we added a check that looks at which way the camera is turned, which stops the panoramic video viewer from updating one side of the canvas when the camera is turned sufficiently to the other side.

5.5 Evaluation

The video controls that we ended up with worked as intended and show a consistent style across the various browsers. Getting the other approaches we tried to work consistently across the plethora of browsers used during this thesis was not possible without significant browser-specific styling and scripting so we decided to use a simple approach for this thesis. The resulting GUI is shown in figure 5.1.

The zoom-based switching between video streams of varying quality resulted in a drastic increase in performance when zoomed out, with the lower quality video only resulting in a small reduction in observable video quality as demonstrated in figure 5.2. Switching from one stream to another is near instant when both files are preloaded or available locally, however this method does involve some latency when streaming both video sources over an internet connection.

The segmentation of the canvas likewise resulted in much better performance whenever the camera was positioned in such a way as to allow us to stop updating one of the video streams, effectively doubling the frame rate This proves that the concept works. Unfortunately the seams of the split are somewhat obvious in our implementation due to color differences that are a result of the cropping process. A higher quality encode, or a more advanced filtering algorithm should address this. Synchronization of the two streams is not currently enforced by our implementation and the streams do desynchronise as the test goes on.

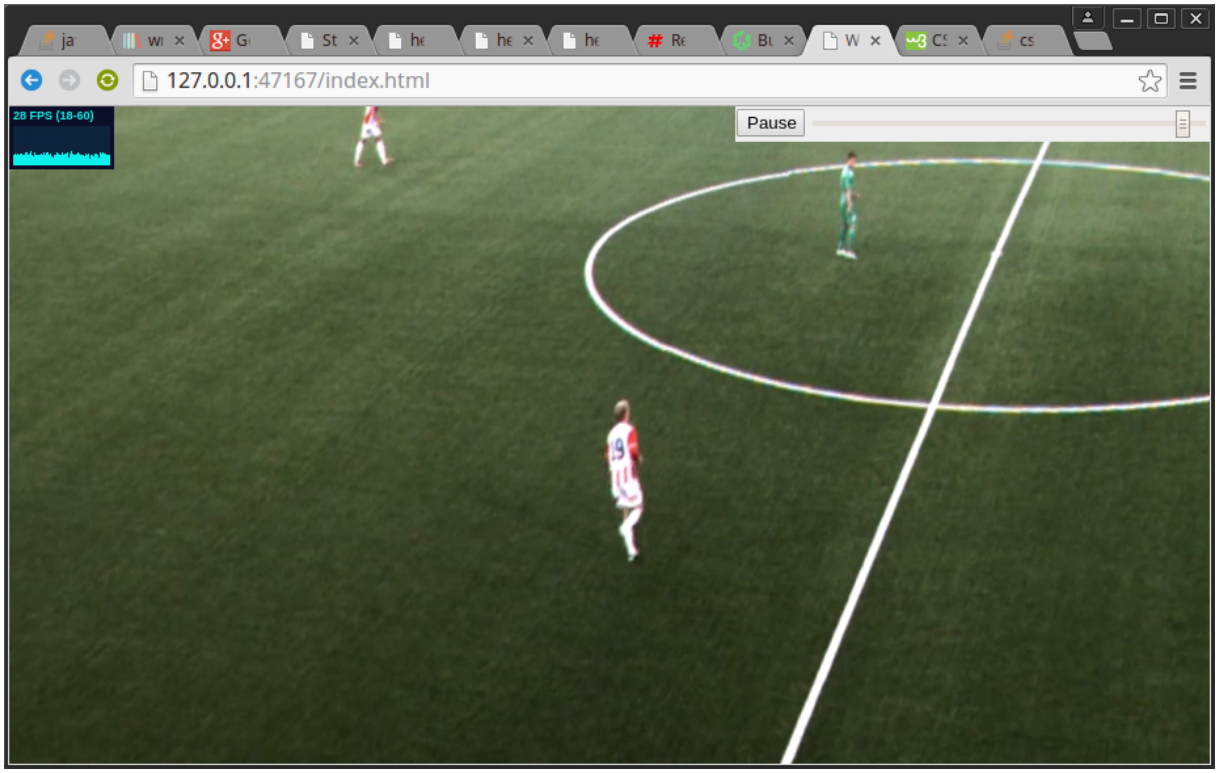


Figure 5.1: Video controls in WebGL-based video panoramic video viewer

(a) High resolution video

(b) Low resoulution video

Figure 5.2: Comparison of low and high video quality streams when zoomed out in WebGL-based panoramic video viewer

5.6 Future Work

A more fully featured GUI should be developed for the WebGL panoramic video viewer. Ideally we would want to integrate the script into a more complete media player where we may switch between a panorama context and a single camera context. This media player should be written in such a way as to dynamically fetch available video streams and should adjust it is own internal workings to accommodate the stream based on included meta-information.

Downloading and drawing large video textures when the current view port of the panoramic video viewer is not displaying is both a waste of resources and a performance concern. A combination of adaptive bit-rate and segmentation by the client could address both those concerns and more work work should be done implementing a fully functional version of this, or some other way of optimizing what data we stream and when. Using Media Source Extensions would allow us to implement a smarter buffering scheme that could reduce the latency when switching from one quality to another.

A more generalized approach to canvas segmentation would allow us to use a higher amount of segments. By balancing the amount of segments rights we could reduce unnecessary rendering when zooming in on any part of the canvas. This would require that more work goes into segmenting video streams in a way that allows us to cleanly combine them afterwards, perhaps by using video segments slightly larger than the canvas segments and blending the seams. Frustum culling is a technique where we could more accurately decide which segments are observable by the camera and which are not. Three.js supports this technique and it should be used in order to make informed decisions on which segments to cull across a wide range of browser window sizes, camera positions and zoom levels.

5.7 Summary

In this chapter we have presented additional improvements that we made to the WebGL-based panoramic video viewer. We introduced the video controls we added to the GUI and two approaches we experimented with in order to increase the performance of the viewer and reduce bandwidth usage. The first involved switching between to different encodes of the same video data based on how far into the video we had zoomed. The second was an idea where we split the video data into two separate videos and played them on separate parts of our canvas geometry. After detailing the implementation process of these topics we discussed the results we achieved, which are promising, but limited to being proof of concepts because of time concerns. We finished the chapter by detailing what the next steps for these new functions might be in the future work section.

Chapter 6

Conclusion

In this chapter we summarize the work that has gone into this thesis, and what we have done to address the problem statement in section 1.2. We will then look at what the main contributions of our thesis were and what future work remains to be done.

6.1 Summary

In this thesis we have improved upon the previous panoramic video viewer prototypes which we described and evaluated in chapter 3. We first looked at the Bagadus system as a whole for context, then we examined the previous work that had gone into developing three generations of panoramic video viewer prototypes. We described the theory behind the panorama de-warping process and how this was implemented in each of the prototypes, before evaluating the results and user experience we got from them. We concluded that while they all worked satisfactory they were difficult to install, unresponsive in use, and required specific hardware to run.

In chapter 4 we presented our WebGL-based prototype which bases itself on a far more intuitive use of a curved canvas, video textures and a perspective camera to achieve the same unwarping effect as that of the previous prototypes while being faster, easier to use and with a series of new features. We detailed how we used JavaScript and the WebGL specification to implement all of the features of the previous prototypes within the context of a simple browser, as well as expanding some of them such as the addition of the gamepad and mouse as control mechanisms. This resulted in a much more accessible platform for the viewing panoramic video captured by the Bagadus system. This was followed by an evaluation of the performance of both WebGL video texture uploads and the WebGL-based panoramic video viewer as a whole on various browsers and computers. Results for WebGL video textures were mixed as browser support is still not optimized for this kind of use, but we nonetheless managed to achieve admirable and useful results on a number of different test machines, which shows that this technology is ready to be used for this purpose, and will only keep getting better as the browser WebGL implementations improve.

Chapter 5 was dedicated to exploring a few possible ways of improving upon our WebGL-based prototype. We specifically discussed how we added video controls to the panoramic video viewer, and how we tried to address the performance and bandwidth concerns caused by working with such large video files. This included a novel method of changing the quality of the video we streamed depending on the level of zoom applied in the panoramic video viewer, as well as a technique where we segmented the curved canvas and only played video in the segments being observed by the camera. This dramatically increased the performance of the viewer and opens up the possibility of greatly reducing the bandwidth used, which makes this new prototype a potential platform for streaming panoramic video of sporting events to large numbers of people.

6.2 Main Contributions

In this thesis we have shown that a full replacement for the current Bagadus panoramic video viewer using open web technologies is desirable and will be achievable in time if not already. Our new prototype

produces adequate results across a wide range of hardware, and works on most browsers. It compares favorably the previous implementations in many areas. The WebGL-based prototype allows users to zoom and pan in high-quality de-warped panoramic videos without installing any additional software. We have shown that by using newly adopted open web standards we can utilize the GPUs and IGPs that most computers are equipped with these days to perform sophisticated multimedia processing in a browser, and that this can be used to simplify the deployment of this part of the Bagadus system. In turn this presents us with the possibility of reaching new audiences with the footage captured by the Bagadus camera array.

Improvements to the GUI and user input means that this new prototype is more responsive than those that came before it. The addition of gamepad and mouse support allows users to make quicker and more precise camera movements that flow more smoothly like a real camera would. The addition of video controls allows users to pause the video and zoom in on specific part of the action, and skip backwards or forwards in time to watch or re-watch specific events.

6.3 Future work

We have discussed some areas of the WebGL-based panoramic video viewer that could be improved in section 4.7. This includes fleshing out the functionality by integrating it with the rest of the Bagadus system and making use of the additional meta-data gathered by the system to provide ball and player tracking, or skipping to identified Bagadus events. OpenCL was brought up as a potential supplement to WebGL should the need for computing intensive calculations arise.

Finding a way of making the panoramic video viewer adapt the canvas depending upon the specific set-up of the Bagadus camera array as well as the aspect ratio of the video stream will be important if we want to support a large number of deployment sites without having to manually configure each one every time a change is made.

In section 5.6 we discussed some additional work that could be done to improve the GUI, performance and bandwidth use of the panoramic video viewer. A fully functional HTML5 video player plug-in with the video panoramic video viewer code as an optional way of presenting video streams would allow us to integrate the solution into a more entertainment-based context. Giving fans the option of customizing their viewing experience of a match they watch online by taking control of the camera could be a valuable feature in the competitive marketplace that is online entertainment.

Additional work must be done to allow the viewer to asynchronously fetch and buffer video data of varying quality in order to provide the viewing experience when there are performance and bandwidth concerns. We discussed a few potential next steps in the effort to address these issues, such as adaptive bit rate streaming and the new Media Source Extension specification. This ties into the big remaining task for the project as a whole, which is integrating it into the Bagadus system. For this to work the viewer must be changed so it can exchange data with the Bagadus back end as we currently rely on manually exporting and feeding video data to the viewer.

Appendix A

Hardware

A.1 Computer Specifications

In this section you can find the specifications of the computers used for testing and development during the writing of this thesis.

A.1.1 Development Machine

Hastur is the primary workstation used for the development of our hardware accelerated panoramic video viewer web applications. It originally came equipped with a legacy GPU which did not support the CUDA technology the earlier prototypes of the panoramic video viewer relied upon, so a modern budget GPU with support for a large variety of GPGPU APIs was installed. The rest of the machine still runs on somewhat outdated hardware and was not deemed representative, which necessitated testing of our prototypes on additional machines described in the next section.

A.1.2 Testing Machines

We chose a variety of computers running on different types of hardware, operating systems and browsers in order to test the performance of our prototypes, and to demonstrate one of the strengths of building this kind of software, namely the cost-effective deployment of cross-platform web applications. The first choice was Azatoth, an enthusiast-level machine equipped with top of the line consumer grade hardware, including the latest generation of discrete GPU. The second is Mi-go, a mid range laptop computer that does not use a discrete GPU, but instead relies on an IGP that is built onto the same die as the CPU. The last test computer is Nodens, an older Macintosh workstation, with a legacy discrete GPU.

Hastur		
CPU	Model	Intel® Core™2 Duo E6750
	Frequency	2.66 GHz
	Logical cores (physical * threads)	2 * 1
GPU	Model	GeForce GTX 750 Ti
	CUDA Compute Capability	5.0
Memory	Model	Kingston KHX6400D2K2
	Amount	4 GB
	Frequency	800 MHz
Storage	Model	Wester Digital HDD WD5000KS
	Amount	500 GB
OS	Name	Linux Mint
	Version	17.2

Table A.1: Hastur specifications

Azatoth		
CPU	Model	Intel® Core™2 Duo E6750
	Frequency	2.66 GHz
	Logical cores (physical * threads)	2 * 1
GPU	Model	GeForce GTX 970
	CUDA Compute Capability	5.0
Memory	Model	Corsair Vengeance Pro
	Amount	8 GB
	Frequency	2400 MHz
Storage	Model	Samsung EVO 840 (SSD)
	Amount	500 GB
OS	Name	Microsoft Windows
	Version	7

Table A.2: Azatoth specifications

Mi-go		
CPU	Model	Intel® Core™ i5 3317U
	Frequency	2.6 GHz
	Logical cores (physical * threads)	2 * 2
GPU	Model	Intel® HD Graphics 4000
	OpenCL version support	1.2
Memory	Model	Generic DDR3
	Amount	4 GB
	Frequency	1600 MHz
Storage	Model	SanDisk U100 (SSD)
	Amount	128 GB
OS	Name	Arch Linux
	Version	4.0.6

Table A.3: Mi-go specification

Nodens		
CPU	Model	Intel® Core™ i7-870
	Frequency	2.93 GHz
	Logical cores (physical * threads)	4 * 2
GPU	Model	ATI Radeon HD 5750
	OpenCL version support	1.2
Memory	Model	Generic DDR3
	Amount	4 GB
	Frequency	1333 MHz
Storage	Model	WDC WD1001FALS (HDD)
	Amount	1 TB
OS	Name	Mac OSX
	Version	10.10.4

Table A.4: Nodens specification

A.2 Graphics Processor Specifications

Listed here are the specifications of the various graphics processors we have used to test the new panoramic video viewer web application described in chapter 4.

GPU	G86	GK106	GM204
Model	GeForce 8500 GT	GeForce GTX 750 Ti	GeForce GTX 970
Base Clock	450 MHz	1020 MHz	1050 MHz
Memory Clock	400 MHz	1350 MHz	3505 MHz
Memory Amount	256 MB	2 GB	4 GB
Memory Bandwidth	12.8 GB/s	86.4 GB/s	224.32 GB/s
OpenGL Version Support	2.1	4.5	4.5
OpenCL Version Support	1.1	1.2	1.2
CUDA Cores	16	240	1664
CUDA Compute Capability	1.1	5.0	5.2

GPU	GT2	RV850
Model	Intel® HD Graphics 4000	ATI Radeon HD 5750
Base Clock	350-1050 MHz	700 MHz
Memory Clock	System shared	1150 MHz
Memory Amount	System shared	1 GB
Memory Bandwidth	25.6 GB/s	73.6 GB/s
OpenGL Version Support	4.0	4.3
OpenCL Version Support	1.2	1.2
CUDA Cores	N/A	N/A
CUDA Compute Capability	N/A	N/A

Table A.5: GPU specifications

Appendix B

Extra tables

Listed here are the full tables of performance data gathered using the online WebGL texture upload test written by Florian Boesch [55]. Each of the computers described in the previous chapter ran the test multiple times with about the same variation you can see in the tables so we have included only the first two passes. These tables are referenced in section 4.6. They include statistics gathered on the time it takes to upload a single frame of video to a GPU texture using WebGL, and how many FPS that effectively results in. The tests are run with multiple video formats, with increasing resolution using both of the two texture upload functions that we may use, with texImage2D being of particular interest to us at this time.

Ideally the time it takes to upload a single frame should be under a millisecond for videos of these sizes, but as long as the FPS remains over 30 we consider that an acceptable result. N/A denotes that the format in question is not supported by the browser being tested.

Codec		H.264		VP8		Theora	
Upload method	Resolution	Frames/s	Time(ms)	Frames/s	Time(ms)	Frames/s	Time (ms)
texImage2D	240p	59.08	3.07	N/A	N/A	N/A	N/A
texImage2D	480p	59.83	4.52	N/A	N/A	N/A	N/A
texImage2D	720p	59.74	9.50	N/A	N/A	N/A	N/A
texImage2D	1080p	42.83	6.01	N/A	N/A	N/A	N/A
texSubImage2D	240p	59.99	3.03	N/A	N/A	N/A	N/A
texSubImage2D	480p	59.97	4.42	N/A	N/A	N/A	N/A
texSubImage2D	720p	59.82	8.91	N/A	N/A	N/A	N/A
texSubImage2D	1080p	29.78	20.44	N/A	N/A	N/A	N/A
texImage2D	240p	59.10	3.06	N/A	N/A	N/A	N/A
texImage2D	480p	59.85	4.50	N/A	N/A	N/A	N/A
texImage2D	720p	59.73	9.51	N/A	N/A	N/A	N/A
texImage2D	1080p	29.75	21.41	N/A	N/A	N/A	N/A
texSubImage2D	240p	60.01	3.05	N/A	N/A	N/A	N/A
texSubImage2D	480p	59.97	4.45	N/A	N/A	N/A	N/A
texSubImage2D	720p	59.66	8.93	N/A	N/A	N/A	N/A
texSubImage2D	1080p	29.79	20.53	N/A	N/A	N/A	N/A

Table B.1: Uploading video textures to Nodens GPU using WebGL (Mac OSX, Safari)

Codec		H.264		VP8		Theora	
Upload method	Resolution	Frames/s	Time(ms)	Frames/s	Time(ms)	Frames/s	Time (ms)
texImage2D	240p	52.83	0.55	54.07	0.55	52.18	0.58
texImage2D	480p	50.80	0.95	45.43	0.94	41.86	0.96
texImage2D	720p	47.82	1.66	49.67	1.55	38.65	1.99
texImage2D	1080p	41.97	1.76	46.37	2.76	44.22	2.97
texSubImage2D	240p	54.58	0.82	47.27	0.86	54.82	0.82
texSubImage2D	480p	51.18	2.58	49.45	2.76	49.39	2.73
texSubImage2D	720p	43.92	6.46	44.70	6.23	43.73	6.37
texSubImage2D	1080p	33.21	14.82	32.12	15.37	31.47	15.58
texImage2D	240p	58.67	0.53	58.75	0.52	58.80	0.52
texImage2D	480p	58.38	0.55	58.01	0.89	58.78	0.54
texImage2D	720p	53.36	1.39	56.37	1.28	57.20	1.26
texImage2D	1080p	54.55	1.24	48.35	2.56	47.29	2.61
texSubImage2D	240p	58.44	0.78	58.76	0.77	58.71	0.76
texSubImage2D	480p	53.24	2.65	53.39	2.60	53.71	2.59
texSubImage2D	720p	43.44	6.52	44.98	6.16	43.17	6.37
texSubImage2D	1080p	33.50	14.76	31.61	15.36	31.40	15.57

Table B.2: Uploading video textures to Nodens GPU using WebGL (Mac OSX, Chrome)

Codec		H.264		VP8		Theora	
Upload method	Resolution	Frames/s	Time(ms)	Frames/s	Time(ms)	Frames/s	Time (ms)
texImage2D	240p	60.00	1.12	60.03	0.53	60.09	0.49
texImage2D	480p	60.01	3.37	60.03	1.80	60.07	1.77
texImage2D	720p	59.97	6.61	60.05	3.85	60.05	3.84
texImage2D	1080p	58.94	13.84	60.02	8.92	59.98	8.84
texSubImage2D	240p	60.00	0.94	60.09	0.26	60.09	0.48
texSubImage2D	480p	60.00	3.41	60.03	1.77	60.07	1.76
texSubImage2D	720p	60.01	6.64	59.98	3.86	60.07	3.86
texSubImage2D	1080p	58.81	13.77	60.05	8.85	60.00	8.87
texImage2D	240p	60.07	0.48	60.07	0.49	60.07	0.49
texImage2D	480p	60.02	1.84	60.09	1.80	60.05	1.81
texImage2D	720p	60.01	4.06	60.06	3.96	60.07	3.95
texImage2D	1080p	58.00	14.04	59.97	9.25	60.00	8.93
texSubImage2D	240p	59.56	0.26	60.09	0.26	60.10	0.48
texSubImage2D	480p	60.08	3.41	60.03	1.79	60.08	1.79
texSubImage2D	720p	60.04	6.72	60.06	3.95	60.07	3.87
texSubImage2D	1080p	57.32	13.88	60.06	9.02	60.00	9.00

Table B.3: Uploading video textures to Azatoth GPU using WebGL (Windows 7, Firefox)

Codec		H.264		VP8		Theora	
Upload method	Resolution	Frames/s	Time(ms)	Frames/s	Time(ms)	Frames/s	Time (ms)
texImage2D	240p	59.07	0.19	59.09	0.37	59.02	0.38
texImage2D	480p	59.09	0.19	59.02	0.48	59.02	0.46
texImage2D	720p	59.03	0.19	59.05	0.76	59.06	0.79
texImage2D	1080p	59.07	0.16	59.06	1.03	59.01	1.30
texSubImage2D	240p	58.94	0.77	58.91	0.43	59.08	0.42
texSubImage2D	480p	59.09	1.94	59.01	1.46	59.08	1.46
texSubImage2D	720p	58.90	4.45	58.99	3.45	58.99	3.54
texSubImage2D	1080p	46.00	11.29	57.40	8.99	57.48	9.13
texImage2D	240p	59.06	0.19	59.07	0.38	59.03	0.38
texImage2D	480p	59.02	0.17	59.10	0.48	59.02	0.48
texImage2D	720p	59.03	0.19	58.83	0.55	59.06	0.78
texImage2D	1080p	59.02	0.15	59.08	1.02	58.98	1.29
texSubImage2D	240p	59.05	0.75	59.06	0.42	59.06	0.41
texSubImage2D	480p	59.06	1.96	59.00	1.46	59.08	1.46
texSubImage2D	720p	58.99	4.39	59.07	3.50	59.06	3.57
texSubImage2D	1080p	45.47	11.28	56.60	9.23	56.53	9.34

Table B.4: Uploading video textures to Azatoth GPU using WebGL (Windows 7, Chrome)

Codec		H.264		VP8		Theora	
Upload method	Resolution	Frames/s	Time(ms)	Frames/s	Time(ms)	Frames/s	Time (ms)
texImage2D	240p	59.37	0.68	59.61	0.71	60.08	0.64
texImage2D	480p	58.78	1.06	59.57	1.17	60.10	1.08
texImage2D	720p	55.68	1.71	58.03	1.55	58.11	1.58
texImage2D	1080p	44.66	3.88	60.09	3.18	49.77	3.49
texSubImage2D	240p	58.92	0.76	59.77	0.73	59.97	0.69
texSubImage2D	480p	56.64	1.06	58.86	1.08	59.28	0.98
texSubImage2D	720p	52.20	1.59	55.93	1.51	56.34	1.50
texSubImage2D	1080p	42.44	3.68	46.74	3.43	47.70	3.42
texImage2D	240p	58.63	0.56	60.00	0.61	59.76	0.57
texImage2D	480p	55.82	0.99	58.34	1.03	58.75	0.90
texImage2D	720p	50.71	1.59	53.98	1.57	54.54	1.49
texImage2D	1080p	39.62	3.72	44.68	3.59	45.28	3.36
texSubImage2D	240p	56.55	0.55	58.83	0.49	59.41	0.51
texSubImage2D	480p	52.67	0.99	55.36	1.05	56.04	0.90
texSubImage2D	720p	47.87	1.56	51.31	1.54	51.66	1.55
texSubImage2D	1080p	44.76	3.95	43.30	3.61	43.80	3.52

Table B.5: Uploading video textures to Hastur GPU using WebGL (Mint Linux, Firefox)

Codec		H.264		VP8		Theora	
Upload method	Resolution	Frames/s	Time(ms)	Frames/s	Time(ms)	Frames/s	Time (ms)
texImage2D	240p	41.51	1.38	43.29	1.28	45.39	1.12
texImage2D	480p	40.44	2.23	41.85	2.10	42.96	1.82
texImage2D	720p	35.03	4.73	38.23	4.18	39.00	3.68
texImage2D	1080p	29.00	10.72	31.60	9.04	31.97	8.09
texSubImage2D	240p	40.55	2.28	42.39	2.20	41.85	2.15
texSubImage2D	480p	35.39	5.37	36.82	4.99	37.74	4.77
texSubImage2D	720p	28.46	12.18	28.94	11.45	28.99	11.27
texSubImage2D	1080p	17.56	28.78	19.74	25.35	20.64	24.32
texImage2D	240p	41.94	1.28	43.11	1.25	42.95	1.13
texImage2D	480p	39.13	2.06	39.54	2.16	41.97	1.65
texImage2D	720p	36.44	3.47	37.13	3.31	38.81	2.96
texImage2D	1080p	27.63	11.61	29.67	9.77	30.00	8.76
texSubImage2D	240p	40.62	2.23	40.27	2.24	40.86	2.05
texSubImage2D	480p	35.70	5.14	36.66	4.95	36.81	4.78
texSubImage2D	720p	27.78	12.23	28.22	11.61	28.75	11.34
texSubImage2D	1080p	17.68	29.64	19.42	25.78	20.01	25.17

Table B.6: Uploading video textures to Hastur GPU using WebGL (Mint Linux, Chrome)

Codec		H.264		VP8		Theora	
Upload method	Resolution	Frames/s	Time(ms)	Frames/s	Time(ms)	Frames/s	Time (ms)
texImage2D	240p	41.52	0.32	37.23	0.38	42.50	0.33
texImage2D	480p	40.00	0.52	41.23	0.53	41.10	0.52
texImage2D	720p	37.45	0.86	39.06	0.81	38.82	0.82
texImage2D	1080p	31.81	1.76	35.00	1.61	35.01	1.62
texSubImage2D	240p	40.17	0.33	40.36	0.33	40.95	0.33
texSubImage2D	480p	38.21	0.54	39.40	0.53	39.61	0.54
texSubImage2D	720p	35.40	0.85	37.27	0.82	37.60	0.81
texSubImage2D	1080p	30.91	1.73	33.49	1.64	33.75	1.62
texImage2D	240p	38.81	0.33	39.48	0.33	39.41	0.34
texImage2D	480p	36.67	0.54	37.82	0.53	37.52	0.55
texImage2D	720p	33.90	0.85	35.91	0.83	35.95	0.83
texImage2D	1080p	29.34	1.75	32.46	1.65	32.56	1.66
texSubImage2D	240p	36.98	0.33	37.37	0.33	38.20	0.33
texSubImage2D	480p	35.50	0.54	36.31	0.53	36.59	0.52
texSubImage2D	720p	32.97	0.86	34.93	0.84	34.59	0.84
texSubImage2D	1080p	35.24	1.71	31.48	1.60	31.37	1.63

Table B.7: Uploading video textures to Mi-go IGP using WebGL (Arch Linux, Firefox)

Codec		H.264		VP8		Theora	
Upload method	Resolution	Frames/s	Time(ms)	Frames/s	Time(ms)	Frames/s	Time (ms)
texImage2D	240p	60.05	1.10	59.99	1.10	60.03	1.05
texImage2D	480p	60.04	1.29	60.04	1.33	60.09	1.26
texImage2D	720p	60.10	2.13	59.96	2.12	60.03	2.02
texImage2D	1080p	58.63	4.66	59.94	3.75	60.07	3.48
texSubImage2D	240p	60.00	1.60	60.05	1.58	60.02	1.57
texSubImage2D	480p	60.03	3.93	60.01	3.88	60.00	3.89
texSubImage2D	720p	59.97	9.78	59.84	9.39	60.00	9.30
texSubImage2D	1080p	36.20	21.93	39.13	20.78	41.31	20.03
texImage2D	240p	60.01	1.12	60.10	1.09	60.03	1.08
texImage2D	480p	60.05	1.14	60.02	1.31	60.05	1.13
texImage2D	720p	60.10	1.79	60.10	1.78	60.08	1.72
texImage2D	1080p	59.41	4.39	59.54	3.80	60.04	3.54
texSubImage2D	240p	60.11	1.61	60.08	1.56	60.08	1.56
texSubImage2D	480p	60.02	3.95	60.02	3.97	60.05	3.87
texSubImage2D	720p	59.99	9.83	59.91	9.34	59.39	9.26
texSubImage2D	1080p	36.91	21.92	39.04	21.02	40.88	20.47

Table B.8: Uploading video textures to Mi-go IGP using WebGL (Arch Linux, Chrome)

Appendix C

Accessing the source code

The source code for the Bagadus system as a whole, including what is described in this thesis, can be located at https://bitbucket.org/mpg_code/bagadussii.

To retrieve the code, run *git clone git@bitbucket.org:mpg_code/bagadusii.git*. The code to the panoramic video implementations described in chapter 3 can then be found in the */vamsi/oldCodes* sub directory, while the code for the WebGL-based prototype may be found in the */mattiahj* sub directory.

Bibliography

[1] Simen Sægrov. Bagadus: next generation sport analysis and multimedia platform using camera array and sensor networks. 2012.

[2] Simon Kuper. *Soccernomics: Why England Loses, Why Spain, Germany, and Brazil Win, and Why the US, Japan, Australia and Even Iraq Are Destined to Become the Kings of the World's Most Popular Sport*. Nation Books, 2014.

[3] Prozone. `http://www.prozonesports.com/`.

[4] Zxy sport tracking. `http://www.zxy.no/`.

[5] Interplay. `http://interplay-sports.info`.

[6] Vamsidhar Reddy Gaddam, Ragnar Langseth, Sigurd Ljødal, Pierre Gurdjos, Vincent Charvillat, Carsten Griwodz, and Pål Halvorsen. Interactive zoom and panning from live panoramic video. In *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*, page 19. ACM, 2014.

[7] Douglas E Comer, David Gries, Michael C Mulder, Allen Tucker, A Joe Turner, Paul R Young, and Peter J Denning. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[8] T Aarnio and M Bourges-Sevenier. Webcl 1.0 specification.

[9] Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Asgeir Mortensen, Ragnar Langseth, Sigurd Ljødal, Østein Landsverk, et al. Bagadus: An integrated real-time system for soccer analytics. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 10(1s):14, 2014.

[10] Pål Halvorsen, Simen Sægrov, Asgeir Mortensen, David KC Kristensen, Alexander Eichhorn, Magnus Stenhaug, Stian Dahl, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Carsten Griwodz, et al. Bagadus: an integrated system for arena sports analytics: a soccer case study. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 48–59. ACM, 2013.

[11] Marius Tennoe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Hakon Kvale Stensland, Vamsidhar Reddy Gaddam, Dag Johansen, Carsten Griwodz, and Pal Halvorsen. Efficient implementation and processing of a real-time panorama video pipeline. In *Multimedia (ISM), 2013 IEEE International Symposium on*, pages 76–83. IEEE, 2013.

[12] The FFMPEG Team. Ffmpeg: A complete, cross-platform solution to record, convert and stream audio and video. `https://www.ffmpeg.org/`. Accessed: 2015-11-7.

[13] Dag Johansen, Magnus Stenhaug, Roger Bruun Asp Hansen, Agnar Christensen, and P-M Hogmo. Muithu: Smaller footprint, potentially larger imprint. In *Digital Information Management (ICDIM), 2012 Seventh International Conference on*, pages 205–214. IEEE, 2012.

[14] Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Carsten Griwodz, Pål Halvorsen, and Dag Johansen. Processing panorama video in real-time. *International Journal of Semantic Computing*, 8(02):209–227, 2014.

[15] Christian J van den Branden Lambrecht. *Vision models and applications to image and video processing*. Springer Science & Business Media, 2001.

[16] Vamsidhar Reddy Gaddam, Ragnar Langseth, Håkon Kvale Stensland, Pierre Gurdjos, Vincent Charvillat, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Be your own cameraman: real-time support for zooming and panning into stored and live panoramic video. In *Proceedings of the 5th ACM Multimedia Systems Conference*, pages 168–171. ACM, 2014.

[17] Svein Arne Pettersen, Dag Johansen, Håvard Johansen, Vegard Berg-Johansen, Vamsidhar Reddy Gaddam, Asgeir Mortensen, Ragnar Langseth, Carsten Griwodz, Håkon Kvale Stensland, and Pål Halvorsen. Soccer video and player position dataset. In *Proceedings of the 5th ACM Multimedia Systems Conference*, pages 18–23. ACM, 2014.

[18] Matthew Brown and David G Lowe. Automatic panoramic image stitching using invariant features. *International journal of computer vision*, 74(1):59–73, 2007.

[19] Ashley Eden, Matthew Uyttendaele, and Richard Szeliski. Seamless image stitching of scenes with large motions and exposure differences. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 2, pages 2498–2505. IEEE, 2006.

[20] Anat Levin, Assaf Zomet, Shmuel Peleg, and Yair Weiss. Seamless image stitching in the gradient domain. In *Computer Vision-ECCV 2004*, pages 377–389. Springer, 2004.

[21] Assaf Zomet, Anat Levin, Shmuel Peleg, and Yair Weiss. Seamless image stitching by minimizing false edges. *Image Processing, IEEE Transactions on*, 15(4):969–977, 2006.

[22] Richard Szeliski. Image alignment and stitching: A tutorial. *Foundations and Trends® in Computer Graphics and Vision*, 2(1):1–104, 2006.

[23] Terrence Fong, Maria Bualat, Matt Deans, Mark Allan, Xavier Bouyssounouse, Michael Broxton, Laurence Edwards, Rick Elphic, Lorenzo Flückiger, Jeremy Frank, et al. Field testing of utility robots for lunar surface operations. In *AIAA Space*, 2008.

[24] Corinna Jacobs. *Interactive panoramas: techniques for digital panoramic photography*, volume 1. Springer Science & Business Media, 2004.

[25] Jonathan Foote and Don Kimber. Flycam: Practical panoramic video and automatic camera control. In *Multimedia and Expo, 2000. ICME 2000. 2000 IEEE International Conference on*, volume 3, pages 1419–1422. IEEE, 2000.

[26] Venkata Peri and Shree K Nayar. Generation of perspective and panoramic video from omnidirectional video. In *Proc. DARPA Image Understanding Workshop*, volume 1, pages 243–245. Citeseer, 1997.

[27] Ulrich Neumann, Thomas Pintaric, and Albert Rizzo. Immersive panoramic video. In *Proceedings of the eighth ACM international conference on Multimedia*, pages 493–494. ACM, 2000.

[28] Itseez. Open source computer vision library. `http://opencv.org`. Accessed: 2015-05-5.

[29] Nvidia. Cuda parallel computing platform. `http://www.nvidia.com/object/cuda_home_new.html`. Accessed: 2013-05-25.

[30] Khronos Group. Open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`. Accessed: 2013-05-25.

[31] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[32] Jon Peddie Research (JPR). Add-in-board market report. http://jonpeddie.com/publications/add-in-board-report. Accessed: 2015-09-7.

[33] Jon Peddie Research (JPR). Press release: Overall gpu shipments dropped 13% in q1'2015 from last quarter. http://jonpeddie.com/press-releases/details/overall-gpu-shipments-dropped-13-in-q12015-from-last-quarter. Accessed: 2015-09-7.

[34] The Papervision3D Team). Papervision3d: Open source realtime 3d engine for flash. http://code.google.com/p/papervision3d/. Accessed: 2015-10-7.

[35] The Away3D Team. Away3d: an open source, real time 3d engine for the flash platform. http://away3d.com/. Accessed: 2015-10-7.

[36] Robert Dawes, Bruce Weir, Chris Pike, Paul Golds, Mark Mann, and Martin Nicholson. Enhancing viewer engagement using biomechanical analysis of sport. *Proceedings of the NEM summit, Istanbul*, pages 16–18, 2012.

[37] Thomas B Moeslund, Graham Thomas, and Adrian Hilton. *Computer Vision in Sports*. Springer, 2014.

[38] Roy Pea, Michael Mills, Joseph Rosen, Kenneth Dauber, Wolfgang Effelsberg, and Eric Hoffert. The diver project: interactive digital video repurposing. *MultiMedia, IEEE*, 11(1):54–61, 2004.

[39] Espen Oldeide Helgedagsrud. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching. 2013.

[40] Dragomir Anguelov, Carole Dulong, Daniel Filip, Christian Frueh, Stéphane Lafon, Richard Lyon, Abhijit Ogale, Luc Vincent, and Josh Weaver. Google street view: Capturing the world at street level. *Computer*, (6):32–38, 2010.

[41] Google. Google maps street view api. http://developers.google.com/maps/documentation/streetview/. Accessed: 2015-11-7.

[42] krpano Gesellschaft mbH. krpano panoramic video support. http://krpano.com/video/. Accessed: 2015-11-7.

[43] Helmut Dersch. Panotools wiki: Panorama viewers. http://wiki.panotools.org/PanoramaViewers. Accessed: 2015-11-7.

[44] Cassio E dos Santos Junior, Jessica Souza, Virginia F Mota, Guilherme S Nascimento, Guilherme S Gorgulho, Arnaldo de A Araujo, et al. Panview: An extensible panoramic video viewer for the web. In *Web Congress (LA-WEB), 2014 9th Latin American*, pages 109–113. IEEE, 2014.

[45] Mike Geraci. Google enters the web video fray with webm. *Interface: The Journal of Education, Community and Values*, 10(5), 2010.

[46] Microsoft Internet Explorer Feedback Program. Webgl video upload to texture not supported. https://connect.microsoft.com/IE/feedbackdetail/view/941984/webgl-video-upload-to-texture-not-supported. Accessed: 2015-22-7.

[47] Scott Graham. Gamepad access in browsers via gamepad api. http://www.gamepadjs.com/. Accessed: 2015-20-7.

[48] Scott Graham, Google, Ted Mielczarek, and Mozilla. Gamepad: W3c working draft 29 april 2015. http://www.w3.org/TR/2015/WD-gamepad-20150429/. Accessed: 2015-20-7.

[49] WebKit Bugzilla. Webgl slow video to texture. https://bugs.webkit.org/show_bug.cgi?id=129626. Accessed: 2015-22-7.

[50] Bugzilla@Mozilla. Enhance render video-to-skiagl performance by gpu-based color space conversion. https://bugzilla.mozilla.org/show_bug.cgi?id=880114. Accessed: 2015-22-7.

[51] Bugzilla@Mozilla. Make webglcontext::teximage2d avoid readback for video elements on windows. https://bugzilla.mozilla.org/show_bug.cgi?id=1060121. Accessed: 2015-22-7.

[52] Chromium Issues. Ehtml5 video to webgl texture upload is slow. https://code.google.com/p/chromium/issues/detail?id=91208. Accessed: 2015-22-7.

[53] Chromium Issues. Webgl, hw video decode: Implement texsubimage2d() for video to webgl. https://code.google.com/p/chromium/issues/detail?id=349871. Accessed: 2015-22-7.

[54] krpano Gesellschaft mbH. Html5 video webgl performance test. http://krpano.com/ios/bugs/ios8-webgl-video-performance/. Accessed: 2015-29-7.

[55] Chromium Issues. Slow video to texture test. http://codeflow.org/issues/slow_video_to_texture/. Accessed: 2015-23-7.

[56] Aditya Mavlankar, Pierpaolo Baccichet, David Varodayan, and Bernd Girod. Optimal slice size for streaming regions of high resolution video with virtual pan/tilt/zoom functionality. In *Proc. of 15th European Signal Processing Conference (EUSIPCO), Poznan, Poland*, 2007.

[57] W. May R. Pantos. Http live streaming, ietf draft. http://tools.ietf.org/html/pantos-http-live-streaming-09.txt. Accessed: 2015-28-7.

[58] Iraj Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *IEEE MultiMedia*, (4):62–67, 2011.

[59] Alex Zambelli. Iis smooth streaming technical overview. *Microsoft Corporation*, 3, 2009.

[60] Aaron Colwell, Adrian Bateman, and Mark Watson. Media source extensions. *W3C Candidate Recommendation*, 2014.